

对一个即将出门旅行的人来说，最需要的是
一张内容详尽、生动、实用的旅行地图。

对即将进入神秘、美丽的C++世界的你，最
需要的是这本C++世界地图册。

本书将带领你畅游整个C++世界，还等什么，
让我们出发吧！

我的第一本 C++书



陈良乔 著

CHINA-PUB.COM

 华中科技大学出版社
<http://www.hustp.com>

内 容 简 介

虽然 C++ 语言纷繁复杂的语法规则让很多学习者望而却步,但是,在本书中,你会发现 C++ 语言的学习也可以如此轻松。本书没有孔乙己式地去深究 C++ 语言的语法细节,也没有重点地去介绍各种高深的 C++ 编程技巧,而是本着简单实用的原则,通俗易懂地向你介绍 C++ 中最重要、最实用的知识。看完本书,你会发现用逻辑控制语句可以把 C++ 语句串珠成链;用函数实际上就是把程序装进一个箱子;当 C++ 语言爱上面向对象思想就有了类与对象;算法就像体育老师,给我们带来一堂别开生面的算法体育课。本书不再是一本枯燥乏味的 C++ 语法介绍书,而是一本通俗易懂的 C++ 故事书,可以让我们在讲故事中轻松学会 C++ 语言。

图书在版编目(CIP)数据

我的第一本 C++ 书/陈良乔 著. —武汉:华中科技大学出版社,2011.5
ISBN 978-7-5609-6995-4

I. 我… II. 陈… III. C 语言-程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2011)第 044235 号

我的第一本 C++ 书

陈良乔 著

策划编辑:陈禹成

责任编辑:陈元玉

封面设计:潘 群

责任校对:周 娟

责任监印:张正林

出版发行:华中科技大学出版社(中国·武汉)

武昌喻家山 邮编:430074 电话:(027)87557437

录 排:华中科技大学惠友文印中心

印 刷:湖北新华印务有限公司

开 本:787mm×1092mm 1/16

印 张:24.5

字 数:608 千字

版 次:2011 年 5 月第 1 版第 1 次印刷

定 价:58.80 元



本书若有印装质量问题,请向出版社营销中心调换
全国免费服务热线:400-6679-118 竭诚为您服务
版权所有 侵权必究

序言

在多年的教学实践中，我深深感到 C++ 语言的灵活和高效，以及 C++ 语言中的面向对象特性能够带给软件开发者无尽想象的空间，同时也深深感到教授 C++ 语言过程中面临的困难和挑战。尽管目前有关 C++ 语言的书籍很多，但学习 C++ 语言仍然让大多数初学者心存畏惧。本书作者结合自己学习和使用 C++ 语言的经验和感悟，尝试用轻松幽默的语言，借助人们熟知的生活概念，形象生动地讲述 C++ 语言。本书对那些渴望掌握 C++ 语言而又心存畏惧的初学者，无疑是一个很好的选择。

作者与我的团队一同快速成长，相信在这个成长的过程中作者也如我一样，经历了从在懵懂中模仿使用 C++ 语言到喜爱 C++ 语言的指针的灵活及面向对象的设计思想。尽管近年来新的计算机语言不断出现，但 C++ 语言因同时具有最贴近计算机数值操作方式的指针操作和最贴近人类社会行为模式的面向对象的特性，而具有其他程序语言无法比拟的强大生命力。可以说，指针和面向对象的特性是 C++ 语言的灵魂，同时也是学习和使用 C++ 的难点。期望那些有所感悟的同行，参与到探寻有效学习 C++ 语言的方法中来。

本书全面阐述了 C++ 语言的基本概念和技术，并且结合了实例进行讲解，另外，还给出了一些公司的典型笔试题，更加突显出其实用性。本书的语言轻松、幽默，经过作者的深入思考和策划使得学习 C++ 不再枯燥。本书的美中不足之处在于讲解还可以进一步深入、个别观点和类比可以进一步完善。如果读者能够结合书后附录中给出的参考书，会收到更好的学习效果。

康 雁

2011 年 3 月 26 日于沈阳

前言

当你拿起本书翻看到这一页的时候，是否在寻找一本既简单实用又通俗易懂的 C++ 参考书？没错，这本书正是你要寻找的最佳 C++ 参考书。

有了这本书的帮助，你的 C++ 学习过程将会是一次愉快的 C++ 世界之旅。用 C++ 编程就是用 C++ 编程语言描述和表达周围现实世界。按照描述和表达现实世界的需要，本书分别介绍了 C++ 语言中最重要的内容，包括基本数据类型、逻辑控制语句、函数、面向对象思想、类与对象，以及标准模板库等相关的知识。学习基本数据类型之后，你会知道如何使用 `int`、`double` 和 `string` 等数据类型来描述现实世界中的数据；学习逻辑控制语句之后，你会知道如何使用 `if...else` 来控制程序逻辑；学习函数之后，你会知道如何用函数来表达完整的算法；而面向对象思想可以帮助我们抽象现实世界；类与对象则将抽象的结果在程序中表达出来；最后的标准模板库，通过提供通用容器来对数据进行管理，通过提供通用算法来对数据进行处理，从而使程序更加简单而优雅。这些内容，都是 C++ 语言中最基本而又最实用的部分，通过这些内容的学习，你完全可以将 C++ 作为自己的语言来描述和表达现实世界。

这么多内容，学习起来困难吗？虽然 C++ 语言纷繁复杂的语法规则让很多学习者望而却步，但是，在本书中，你会发现 C++ 语言的学习也可以如此轻松。本书没有孔乙己式地去深究 C++ 语言的语法细节，也没有重点地去介绍各种高深的 C++ 编程技巧，而是本着简单实用的原则，通俗易懂地向你介绍 C++ 中最重要、最实用的知识。看完本书，你会发现用逻辑控制语句可以把 C++ 语句串珠成链；用函数实际上就是把程序装进一个箱子；当 C++ 语言爱上面向对象思想就有了类与对象；算法就像体育老师，给我们带来一堂别开生面的算法体育课。本书不再是一本枯燥乏味的 C++ 语法介绍书，而是一本通俗易懂的 C++ 故事书，可以让我们在讲故事中轻松学会 C++ 语言。

本书不仅仅是一本入门参考书，它更是一本关于 C++ 编程经验总结的书。本人使用 C++ 编程已有 10 多年时间，同时多年担任微软最有价值专家（Microsoft Most Valuable Professional, MVP），接触到很多来自实践的问题，也积累了丰富的实践经验。我的这些宝贵实践经验都融入了本书中。通过本书，你不仅可以获得关于 C++ 语言的知识，更重要的，你可以获得这些宝贵的实践经验。例如，表示浮点数的 `float` 和 `double`，我们到底该如何选择？为什么 `const` 关键字这么重要？指针和引用的区别到底在哪里？这些都是来自实践的问题，在

你读完这本书后，你将获得完美的答案，这些答案会帮助你从一个 C++ 新手成长为一个经验丰富的 C++ 开发人员。如果你已经在进行 C++ 程序开发，也可以通过这些经验的积累，让自己的 C++ 编程技能更上一层楼。

最后，当你在 C++ 学习之旅中累了、倦了，或者遇到问题的时候，不妨来我的“有间客栈”坐坐，链接地址：<http://imcc.blogbus.com/>。在这里，我可以为你答疑解惑，让你的 C++ 学习之旅更加轻松惬意。

还等什么呢？即刻拿起这本书，开始我们的 C++ 学习之旅吧！

致谢

本书从最初提出设想到最终成书经历了大约四年时间，我希望能四年磨一剑，能给大家奉上一本好书，一本有价值的书。

在这四年的写作过程中，我获得了很多人的帮助，最终才有了大家手中的这本书。在这里，我要感谢父母对我的默默支持；感谢亲爱的贾玮，你的支持、鼓励和期许是我的动力和灵感的源泉；感谢为本书作序的康雁老师，我的好老师；感谢周静姐姐、唐总和兰姐给予我的教诲；感谢好朋友丁春利、马洪旭在我最困难的时候给予我的无私帮助；感谢 Elmar Driesch 先生、Andreas Muench 先生及 Thorsten Thomsen 先生给予我的指导；感谢本书的编辑，从最初的陈禹成编辑到后来的徐定翔、陈元玉编辑，是你们成就了这本书；感谢所有关心我的朋友，我爱你们！

陈良乔

2011 年 2 月于西安

目录

第 1 篇 叩开 C++ 世界的大门	1
第 1 章 C++ 世界地图	3
1.1 C++ 是什么	3
1.2 C++ 的前世今生	3
1.2.1 从 B 到 C	4
1.2.2 从 C 到 C++	4
1.2.3 从 C++ 到 .NET Framework 的 CLI	6
1.2.4 最新标准 C++0x 让 C++ 重新焕发活力	8
1.2.5 C++ 和 C# 不得不说的点儿事儿	8
1.2.6 C++ 世界的五大子语言	9
1.3 C++ 世界版图	10
1.3.1 Windows 系统下的 C++ 开发	10
1.3.2 Linux 系统下的 C++ 开发	11
1.3.3 嵌入式系统下的 C++ 开发	11
1.4 如何学好 C++	12
1.4.1 将自然语言转换为 C++ 程序设计语言	12
1.4.2 “多读多写”是学好 C++ 的不二法门	14
1.4.3 和 Google 做朋友	14
第 2 章 与 C++ 第一次亲密接触	17
2.1 一个 C++ 程序的自白	17
2.1.1 用 Visual Studio 创建 C++ 程序	17
2.1.2 以手工方式创建 C++ 程序	20
2.1.3 C++ 程序=预编译指令+程序代码+注释	21
2.1.4 编译器和链接器	25
2.1.5 C++ 程序的执行过程	26
2.1.6 程序的两大任务：描述数据与处理数据	27

2.2	基本输入/输出流	28
2.2.1	标准的输入和输出对象	29
2.2.2	输出格式控制	31
2.2.3	读/写文件	32
2.3	最常用的开发环境 Visual Studio	34
2.3.1	Visual C++的常用菜单	35
2.3.2	Visual C++的常用视图	44
2.4	C++世界旅行必备的物品	46
2.4.1	编程助手 Visual Assist	46
2.4.2	代码配置管理工具 Visual Source Safe	48
2.4.3	CodeProject 和 CodeGuru	48
2.4.4	C++百科全书 MSDN	48
第 2 篇	欢迎来到 C++ 世界	51
第 3 章	C++ 世界众生相	53
3.1	C++ 中的数据类型	53
3.2	变量和常量	54
3.2.1	声明变量	55
3.2.2	给变量取个好名字	55
3.2.3	变量初始化	57
3.2.4	常量	57
3.2.5	用宏与 const 关键字定义常量	59
3.3	数值类型	62
3.3.1	整型数值类型	62
3.3.2	浮点型数值类型	63
3.4	布尔类型	64
3.5	字符串类型	65
3.5.1	字符类型	65
3.5.2	字符串类型	66
3.6	数组	67
3.6.1	数组的声明与初始化	68
3.6.2	数组的使用	69
3.7	枚举类型	71

3.8	用结构体类型描述复杂的事物	73
3.8.1	结构体的定义	73
3.8.2	结构体的使用	74
3.9	指向内存位置的指针	76
3.9.1	指针就是表示内存地址的数据类型	76
3.9.2	指针变量的定义	77
3.9.3	指针的赋值和使用	78
第 4 章	将语句编织成程序	81
4.1	用运算符对数据进行运算	81
4.1.1	用表达式表达设计意图	82
4.1.2	算术运算符	82
4.1.3	赋值操作符	84
4.1.4	关系运算符	84
4.1.5	逻辑运算符	86
4.1.6	运算符之间的优先顺序	87
4.1.7	将表达式组织成语句	89
4.2	条件选择语句	90
4.2.1	if 语句	90
4.2.2	并列选择的 switch 语句	93
4.3	循环控制语句	97
4.3.1	while 循环	97
4.3.2	do...while 循环	99
4.3.3	for 循环	100
4.3.4	循环控制: break 和 continue	102
4.4	从语句到程序	104
4.4.1	程序是控制语句串联起来的语句	104
4.4.2	豪华工资统计程序	106
第 5 章	用函数封装程序功能	109
5.1	函数就是一个大“箱子”	109
5.1.1	函数的声明和定义	110
5.1.2	函数调用机制	113
5.1.3	函数的声明与函数调用	117
5.1.4	函数参数的传递	119

5.1.5	函数的返回值.....	121
5.2	内联函数.....	123
5.2.1	用体积换速度的内联函数.....	123
5.2.2	内联函数的使用规则.....	124
5.3	重载函数.....	125
5.3.1	重载函数的声明.....	125
5.3.2	重载函数的解析.....	128
5.4	函数设计的基本规则.....	129
5.4.1	函数声明的设计规则.....	129
5.4.2	函数体的设计规则.....	131
第 6 章	当 C++ 爱上面向对象.....	135
6.1	从结构化设计到面向对象程序设计.....	135
6.1.1	“自顶向下，逐步求精”的结构化程序设计.....	136
6.1.2	面向对象程序设计.....	137
6.1.3	面向对象的三座基石：封装、继承与多态.....	138
6.2	类：当 C++ 爱上面向对象.....	142
6.2.1	类的声明和定义.....	142
6.2.2	使用类创建对象.....	146
6.2.3	构造函数和析构函数.....	148
6.2.4	拷贝构造函数.....	152
6.2.5	操作符重载.....	155
6.2.6	类成员的访问控制.....	157
6.2.7	在友元中访问类的隐藏信息.....	160
6.3	类如何面向对象.....	162
6.3.1	用类机制实现封装.....	163
6.3.2	用基类和派生类实现继承.....	164
6.3.3	用虚函数实现多态.....	171
6.4	实战面向对象：工资管理系统.....	175
6.4.1	从问题描述中发现对象.....	175
6.4.2	分析对象的属性和行为.....	176
6.4.3	实现类的属性和行为.....	177
6.5	高手是这样炼成的.....	183
6.5.1	C++ 类对象的内存模型.....	183

6.5.2 指向自身的 this 指针	185
第 7 章 C++世界的奇人异事	189
7.1 一切指针都是纸老虎：彻底理解指针	189
7.1.1 指针的运算	189
7.1.2 灵活的 void 类型和 void 类型指针	192
7.1.3 指向指针的指针	194
7.1.4 指针在函数中的应用	195
7.1.5 引用	198
7.2 程序中的异常处理	203
7.2.1 异常处理	203
7.2.2 异常的函数接口声明	206
7.2.3 合理使用异常处理	207
7.3 编写更复杂的 C++程序	208
7.3.1 源文件和头文件	208
7.3.2 名字空间	210
7.3.3 作用域与可见性	214
7.3.4 编译预处理	218
7.4 高手是这样炼成的	220
7.4.1 用宏定义化繁为简	220
7.4.2 用 typedef 定义类型的别名	221
7.4.3 用 const 保护数据	223
第 3 篇 攀登 C++世界的高峰	227
第 8 章 用 STL 优雅你的程序	229
8.1 跟 STL 做朋友	230
8.1.1 算法 + 容器 + 迭代器 = STL	230
8.1.2 在程序中使用 STL	231
8.1.3 STL 到底好在哪里	233
8.2 用模板实现通用算法	234
8.2.1 函数模板	235
8.2.2 类模板	238
8.2.3 模板的实例化	240
8.2.4 用模板实现通用算法	241

第 9 章	STL 中的容器管理数据	245
9.1	容器就是 STL 中的瓶瓶罐罐	245
9.1.1	操作容器中的数据元素	247
9.1.2	使用迭代器访问容器中的数据元素	247
9.1.3	容器的使用说明书	249
9.1.4	如何选择合适的容器	252
9.2	vector 容器是数组的最佳替代者	253
9.2.1	创建并初始化 vector 对象	253
9.2.2	vector 容器的操作	256
9.2.3	访问 vector 容器中的数据	257
9.3	可以保存键值对的 map 容器	258
9.3.1	创建并初始化 map 容器	258
9.3.2	将数据保存到 map 容器中	259
9.3.3	根据键找到对应的值	260
第 10 章	用 STL 中的通用算法处理数据	263
10.1	STL 算法中的“四大帮派”	263
10.2	容器元素的查找与遍历	264
10.2.1	用 for_each() 算法遍历容器中的数据元素	264
10.2.2	用 find() 和 find_if() 算法实现线性查找	266
10.3	容器元素的复制与变换	270
10.3.1	复制容器元素: copy() 算法	270
10.3.2	合并容器元素: merge() 算法	272
10.3.3	变换容器元素: transform 函数	274
10.4	容器元素的排序	276
10.4.1	使用 sort() 算法对容器中的数据进行排序	276
10.4.2	对排序的规则进行自定义	279
10.5	实战 STL 算法	282
10.5.1	“算法”老师带来的一堂别开生面的体育课	282
10.5.2	删除容器中的冗余元素	284
第 11 章	函数指针、函数对象与 Lambda 表达式	287
11.1	函数指针	287
11.1.1	函数指针的声明与赋值	287
11.1.2	用函数指针调用函数	289

11.1.3	用函数指针实现回调函数.....	291
11.1.4	将函数指针应用到 STL 算法中.....	293
11.2	函数对象.....	295
11.2.1	定义一个函数对象.....	295
11.2.2	利用函数对象记住状态数据.....	297
11.3	用 Lambda 表达式编写更简单的函数.....	299
11.3.1	最简单直接的函数表达形式: Lambda 表达式.....	299
11.3.2	Lambda 表达式的语法规则.....	301
11.3.3	Lambda 表达式的复用.....	302
第 12 章	C++世界的几件新鲜事.....	305
12.1	用右值引用榨干 C++的性能.....	305
12.1.1	什么是右值.....	305
12.1.2	右值引用在函数返回值上的应用.....	306
12.1.3	STL 算法中被浪费的右值.....	307
12.1.4	右值引用如何提高性能.....	310
12.2	智能指针 shared_ptr.....	312
12.2.1	C++的内存管理.....	312
12.2.2	用聪明的 shared_ptr 解决内存管理问题.....	313
12.2.3	智能指针的应用场景.....	315
12.2.4	shared_ptr 的使用.....	316
12.2.5	shared_ptr 与标准库容器.....	321
12.2.6	对 shared_ptr 进行自定义.....	323
12.3	用 PPL 进行多线程开发.....	326
12.3.1	多核给程序设计带来的挑战.....	326
12.3.2	PPL 带来免费的午餐.....	327
12.3.3	PPL 中的并行算法.....	329
12.3.4	PPL 中的并行任务.....	331
12.3.5	PPL 中的并行对象和并行容器.....	334
12.3.6	PPL 之外的另一种选择: OpenMP.....	340
第 13 章	找工作就靠它了.....	343
13.1	打好基础.....	343
13.1.1	基本概念.....	343
13.1.2	函数.....	346

13.1.3 面向对象思想.....	348
13.1.4 类与对象.....	349
13.1.5 STL	360
13.2 积累经验	362
13.3 考查智力	364
附录 A 接下来该读什么书	367
A.1 开山鼻祖：《C++程序设计语言》	367
A.2 初学者必看：《C++ Primer 中文版（第4版）》	368
A.3 百科全书：《代码大全，第2版》	368
A.4 内功秘籍：《Effective C++（Third Edition）》	370
A.5 经验很重要：《C++编程规范》	370
后记.....	373



第1篇

叩开〇++世界的大门



C++世界地图

对一个即将到陌生地方去旅行的人来说，什么是最重要的和必需的？没错，是一张内容丰富详尽、生动有趣的旅行地图。借助这张地图，就知道在什么地方停车吃饭、在什么地方打尖住店。即将进入陌生 C++ 世界的各位旅行者对 C++ 世界有太多的问题和疑惑。C++ 是什么？C++ 是怎么来的？C++ 能做什么？如何学好 C++？

面对这些问题，同样需要一张 C++ 世界地图。这张 C++ 世界地图可以解答这些问题和疑惑，让我们清晰地认识 C++ 世界。同时，我们可以通过这张 C++ 世界地图，了解 C++ 世界的整个面貌：有哪些好玩的地方，有哪些有趣的故事，有哪些有用的知识，有哪些危险而需要注意的地方。这张 C++ 世界地图，将带领我们畅游整个 C++ 世界。

还等什么，让我们出发吧！

1.1 C++是什么

C++ 是什么？这几乎是每个即将进入 C++ 世界的旅行者要问的第一个问题。面对这个问题，一千个人有一千种答案。有人说，C++ 是一种高级程序设计语言；有人说，C++ 是 C 语言的继承者；有人说，C++ 是一种面向对象的思维方式；有人说，C++ 很好很强大；更有人说，不要迷恋 C++，C++ 只是一个传说。可以说，这些答案都从某个侧面回答了“C++ 是什么”，但是又都不全面。

在这里，我不能、也不会给你所谓的完美答案。我能够做到的，就是给你描绘美好的 C++ 世界，讲述其中的各色公民、风俗见闻，为各位旅行者指引方向和路径。等各位完成这奇妙的 C++ 世界之旅之后，自然会有自己的答案。

从现在开始，寻找答案吧！

1.2 C++的前世今生

读史可以使人明智。

C++作为一门高级程序设计语言，可说是历史悠久，算得上是程序设计语言中的“老革命”了。了解C++的发展历史，可以加深我们对这门语言的认识，了解C++的本质内涵，了解C++的文化，从而更好地学习和掌握这门语言。

传说，很久很久以前……

1.2.1 从B到C

1967年，著名的计算机科学家丹尼斯·里奇（Dennis Ritchie）进入美国AT&T的贝尔实验室工作。一开始，里奇和他的同事肯·汤普森（Ken Thompson）开始研究DEC PDP-7机器，但是他们发现这个机器上写程序很困难，只能用汇编语言编程。于是汤普森设计了一种高级程序语言，命名为B语言。但是B语言本身设计的缺陷使汤普森在内存的限制面前一筹莫展。到了1973年，里奇对B语言进行改良，他赋予这门新语言强有力的系统控制方面的能力。新语言非常简洁、高效，里奇把它命名为C语言，意为B语言的下一代程序设计语言。

知道更多：B语言从哪里来

C语言来自B语言，那么B语言是不是来自A语言呢？B语言之前并不存在A语言，之所以取名B语言，是为了纪念作者的妻子，他妻子名字的第一个字母是B。

嗯，程序员中也有情圣啊！

1978年，里奇和布朗一起出版了著名的《C Programming Language》一书，C语言随后成为世界上应用最广泛的高级程序设计语言，这个版本的C语言被称为K&R C。1989年，C语言被ANSI标准化（ANSI X3.159—1989）。标准化的目的是扩展K&R C，加入新的特性。在K&R C发布后，又不断有人为C语言添加新特性，但C语言的标准在一段相当长的时间内都保持不变，直到20世纪90年代，标准才被更新，这就是ISO 9899:1999（1999年发布）。这个版本就是通常提及的C99。ANSI于2000年3月采用了这个新标准。

1.2.2 从C到C++

语言的发展是一个逐步递进的过程。1979年4月，同样来自贝尔实验室的本贾尼·斯特劳斯特卢普（Bjarne Stroustrup）博士与同事接受一项工作，尝试分析UNIX的内核。当时没有合适的工具能够有效地分析内核分布形成的网络流量，将内核模块化，所以斯特劳斯特卢普博士的工作进展很慢。同年10月，斯特劳斯特卢普博士设计了一个可以运行的预处理程序，称之为“Cpre”，它为C语言加上了类似Simula语言的类机制。在这个过程中，斯特劳斯特卢普博士产生了创建一门新语言的想法。贝尔实验室对这个想法很感兴趣，就让斯特劳斯特

卢普博士等人组成一个开发小组，专门进行研究。

当时这门新语言不是叫 C++，而是叫 C with class，它只是 C 语言的有效扩充，后来才更名为 C++。当时 C 语言已经在所有程序设计语言中居于老大的地位，要想发展一种新的语言，最强大的竞争对手就是 C 语言。C++ 当时面临两个挑战：第一，C++ 要在运行时间、代码紧凑性和数据紧凑性方面与 C 语言相媲美；第二，C++ 要尽量避免在语言应用领域的限制。在这种情况下，最简单的方法是继承 C 语言的一些特性，让 C++ 语言具备 C 语言的各种优点。斯特劳斯特卢普博士为了突破 C 语言的种种局限，借鉴了其他程序设计语言的优点，实践了编程界由来已久的“拿来主义”。例如：C++ 从 Simula 继承了类的概念；从 Algol68 继承了操作符重载、引用以及在任何地方声明变量的能力；从 BCPL 获得了“//”注释；从 Ada 得到了模板、名字空间；从 Ada、Clu 和 ML 取来了异常处理等。通过这一系列动作，C++ 具备了多种程序设计语言的优秀基因，既系出名门，又博采众家之长，从而完成了从 C 到 C++ 的进化。

其后，C++ 又经历了长期的发展，随着标准模板库 (Standard Template Library, STL) 的出现、泛型编程的发展，C++ 在 2000 年左右出现了发展史上的一个高峰，成为程序设计语言中的无冕之王。

知道更多：C++大事记

1983 年 8 月，C++ 首次投入使用，开天辟地。

1983 年 12 月，Rick Mascitti 建议将 C with class 更名为 CPlusPlus，亦即 C++。C++ 从此名正言顺。同年，C++ 吸收了很多新的特性，其中包括虚函数、函数名、操作符重载、常数、用户可控制的自由空间储存区、改良的类型检查及新的双斜线“//”单行注释风格。

1985 年 2 月，C++ Release 1.0 发布。

1985 年 10 月，斯特劳斯特卢普博士完成了经典巨著《The C++ Programming Language》的第一版。

1989 年，C++ Release 2.0 发布。它引入了多重继承、抽象类、静态成员函数、常数成员函数及成员保护等新特性。C++ 中面向对象的思想更加成熟。

1990 年 3 月，第一次 ANSI X3J16 技术会议在新泽西州召开。

1990 年 7 月，C++ 加入模板。

1990 年 11 月，C++ 加入异常处理。

1991 年 6 月，《The C++ Programming Language》第二版完成。

1991 年 6 月，第一次 ISO WG21 会议在瑞典召开。

1993 年 3 月，在俄勒冈州加入运行时类型识别。

1993 年 7 月，在德国慕尼黑加入名字空间。

1994 年 8 月，ANSI/ISO 委员会草案登记。

1997 年 7 月，《The C++ Programming Language》第三版完成。

1998 年 10 月，ISO 标准通过表决被接受。

1998 年 11 月，ISO 标准得到批准。同年，C++0x 公开，它是目前计划中的 C++ 编程语言的新标准，将取代现行的 C++ 标准 ISO/IEC 14882。

2003 年，在官方公布 1998 标准的 5 年之后，C++ 标准委员会处理缺陷报告，并于 2003 年发布了一个 C++ 标准的修正版本，称为 C++03。新的标准包含了核心语言的新功能，同时扩展了 C++ 标准程序库，合并了大部分的 C++ Technical Report 1 程序库。

2005 年，公布一份名为 Library Technical Report 1（简称 TR1）的技术报告。虽然它不属于官方标准，但它所提出的几个扩展建议有望成为新 C++ 标准的一部分。目前，几乎所有流行的 C++ 编译器都已经支持 TR1。

2008 年 10 月，C++0x 的最新报告 N2800 公开。

到今天为止，C++ 的最新标准 C++0x 已经准备完成，正在等待最后的通过并公布。

1.2.3 从 C++ 到 .NET Framework 的 CLI

微软于 2000 年推出的 .NET Framework，是支持快速开发、部署网站服务及应用程序的开发平台。这个框架有两个目标：第一个目标是提高 Windows 操作系统平台应用程序的开发效率，特别是改善组件对象模型（Component Object Model, COM）的开发；第二个目标是提供支持发展软件服务（software service）的开发平台。

为了使 C++ 应对新的开发趋势，在这个面向未来的开发平台上占有一席之地，微软把 C++ 改造成 .NET Framework 中的 C++/CLI（Common Language Infrastructure，公共语言结构），代替 C++ 托管扩展，从而允许大量熟悉 C++ 的开发人员可以继续 .NET Framework 平台上开发 C++ 应用。

这门语言在兼容原有的 C++ 标准的同时，简化了托管代码扩展的语法，提供了更好的代码可读性和易用性。微软向 ECMA 提交了 C++/CLI 的标准化请求，使其通过 ECMA 成为正式的标准。C++/CLI 现在可以被 Visual C++ 2005 和更高版本的编译器支持。C++/CLI 的部分特性已经申请了专利。

到底什么是 C++/CLI？它跟传统的 C++ 又有什么不同呢？

CLI 指的是通用语言结构，一种支持动态组件编程模型的多重结构。在整个 CLI 中，最重要的是公共语言运行时（Common Language Runtime, CLR），它负责管理微软中间语言（Microsoft Intermediate Language, MSIL）代码的运行环境。CLR 位于 CLI 的下半部分（如图 1-1 所示），主要包括类加载器（class loader）、实时编译器（IL to native compilers）和一个运行时环境的垃圾收集器（garbage collector）。CLI 运行在底层操作系统与程序之间，为 MSIL 代码提供运行的环境，这使得 CLI 成为一个实时的软件层、一个有效的执行系统。可以将任何语言编写的代码通过特定的编译器转换为 MSIL 代码，然后在 CLI 上运行。

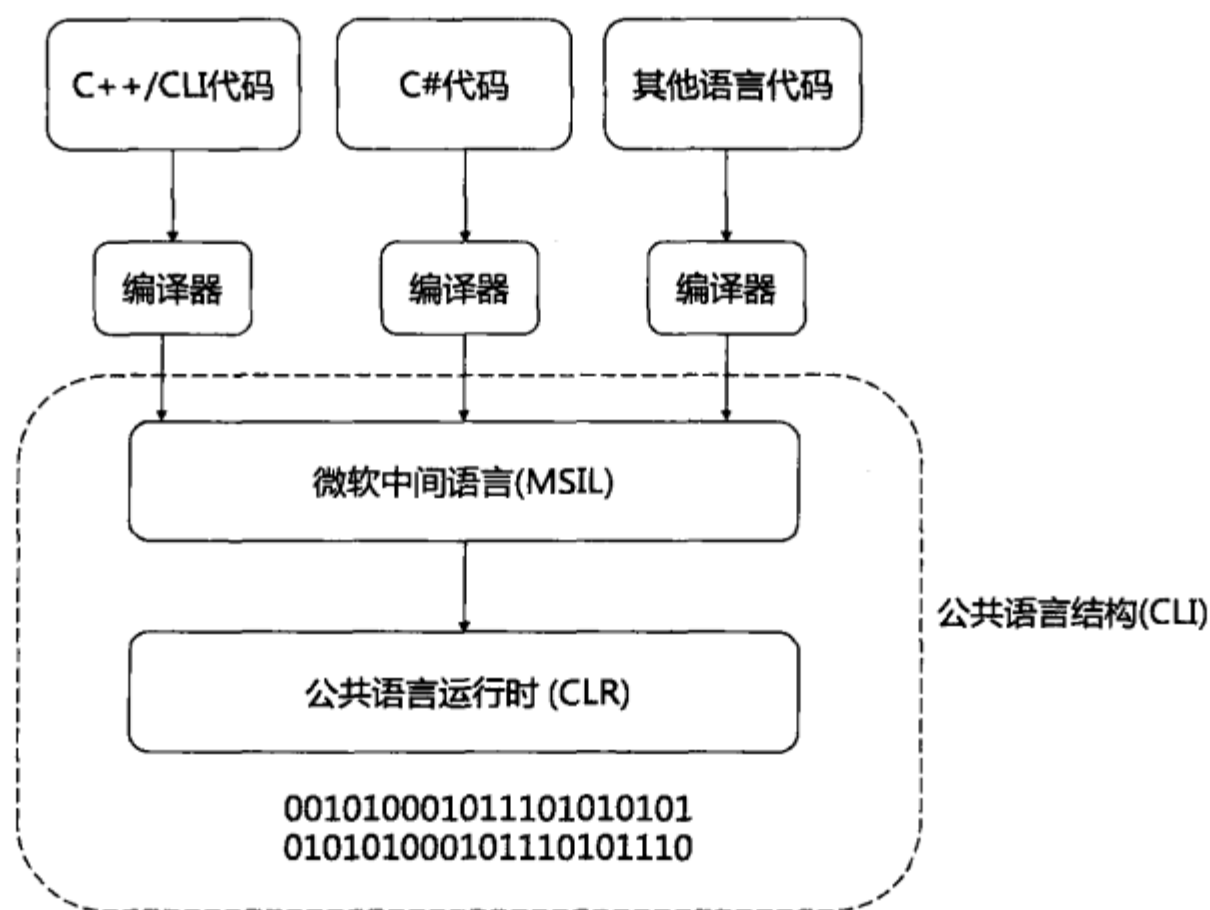


图 1-1 C++/CLI 的结构

C++和 CLI 结合起来就成了一种可以经过特殊的编译器编译之后运行在 CLI 之上的 C++ 语言。斜杠“/”代表 C++和 CLI 的捆绑，这个捆绑使得 C++/CLI 同时具备了 C++和 CLI 这两个方面的特性。首先，C++/CLI 继承了 C++的大部分语法规则，使得我们可以轻松地将 C++代码转换为 C++/CLI 代码。开发人员可以充分利用已有的 C++编码经验，使用 C++/CLI 为 .NET Framework 平台开发新的应用程序。其次，当我们用 C++/CLI 编写的托管代码被运行时，代码将被 CLR 所管理，它提供了诸如垃圾收集等现代高级程序设计语言的特性，同时也实现了 C++/CLI 与 .NET Framework 支持的其他语言之间的互操作性，让我们可以通过 C++/CLI 使用 .NET Framework 平台上丰富的组件来提高开发效率。可以说，借助强大的 .NET Framework，C++/CLI 使得 C++这门“古老”的程序设计语言做到与时俱进，能够开发面向未来的丰富应用。

1.2.4 最新标准 C++0x 让 C++重新焕发活力

自从斯特劳斯特卢普博士发明并实现了 C++语言之后，在面向对象语言迅速发展的时代背景下，C++以其面向对象的语言特性、对 C 语言的良好兼容，以及其接近 C 语言的性能效率，在工业界占据了相当大的份额，成为程序设计语言中的无冕之王。在其后的发展中，C++ 又不断引入新的内容。标准模板库和 Boost 程序库的出现、泛型程序设计的流行，使得 C++ 牢牢占据了 TIOBE 编程语言排行榜前三名的位置，成为业界最流行的程序设计语言之一。

随着硬件技术的不断发展，以及 Java、C#等新语言的涌现，C++的发展受到了很大的冲击，在业界的应用范围不断萎缩。C++曾经是 Visual Studio 6.0 中的首选语言，但是在后继版本的 Visual Studio 中，特别是在微软推出 .NET Framework 之后，C++的地位不断下滑，被后来居上的 C#抢了风头。很多钟情于 C++的程序员不禁发出这样的感叹：“C++老矣，尚能编否？”

虽然 C++在发展历程中经历了上述小小的波折，但是应当看到，世界上还有无数的 C++ 代码在稳定地运行着，这些代码还需要维护和升级。另外，C++在某些领域还具有不可替代的优势，无数基于 C++的新项目正在进行着。为了应对现代程序设计语言的发展及业界的需求，C++也积极汲取现代程序设计语言的精华，C++的新标准 C++0x 正是在这种背景之下应运而生的。这些新特性包括 Lambda 表达式、智能指针 `shared_ptr`、`auto` 关键字、右值引用、多任务内存模型等。这些新特性的引入，进一步增强了 C++在性能方面的优势，同时也改善了 C++的可用性，使得 C++成为了一门“又快又好”的程序设计语言。这些新特性给 C++ 注入了新的活力，使得 C++重新焕发青春，带来 C++的复兴。

1.2.5 C++和 C#不得不说的点事儿

自从微软推出全新的开发语言 C#之后，关于 C++与 C#之间的争论就没有停止过。就像 C++继承了 C 语言的许多特性一样，C#也继承了 C++的许多特性，同时增加了很多现代编程语言的新特性。配合强大的 .NET Framework，C#下的应用开发越来越简单，应用也越来越广泛。C++会不会被新兴的 C#革命？我们应该学习 C++还是 C#？

圣经上说：你必须知道真相，真相会使你自由。虽然 C#和 .NET Framework 让开发变得更简单，使用几行 C#代码就可以完成几十行 C++代码才能完成的功能，而且 C#具有简单的类库操作和面向对象编程的完美特性，但是，所谓成也萧何，败也萧何，正是因为 .NET Framework 的引入，在 C#和操作系统之间隔了一层，让我们无法了解 C#背后的真相，处处受制于 .NET Framework。同样实现一个功能，使用 C#我们只有一种方法，而使用 C++，我们如果明白背后的机制，就可以用不同的方法应对不同的情况，实现最优的方案。没有编程语言比 C++更加贴近 Windows 操作系统了，这一点是不可否认的。可以说，只要追求自由的人存在，C++就不会消失；只要操作系统是用 C++写的，C++就不会消失。

从应用领域上讲，C#主要应用在 Windows 平台上，用于开发与用户界面、网络和数据库相关的应用。而 C++主要应用在 Windows、Linux 和嵌入式系统等平台，其业务领域也非常广泛，从服务器应用程序的开发到多媒体游戏的开发，从图像处理到工程控制，处处都有 C++的身影。平台的广泛性让 C++的应用范围更加广泛。

C#是继承自 C++的，学好 C++之后，可以轻松地学好 C#；但是，学好 C#却不一定能保证学好 C++。

语言无所谓好坏强弱之分，C#能做的，C++不一定都能做，而 C++能做的，C#也不一定都能做好。所以，根据应用场景选择合适的语言才是最重要的。在自由和束缚之间，我们选择自由；在 C++和 C#之间，我选择 C++。

1.2.6 C++世界的五大子语言

从 1983 年首次投入使用至今，C++已经有 40 多年的发展历史了，在发展过程中，不同的应用领域、不同的开发思想形成了不同的 C++子语言。每个子语言各有所长，就像 C++世界的五岳剑派，各自在自己的领域独领风骚，形成 C++世界百花齐放的繁荣局面，如图 1-2 所示。

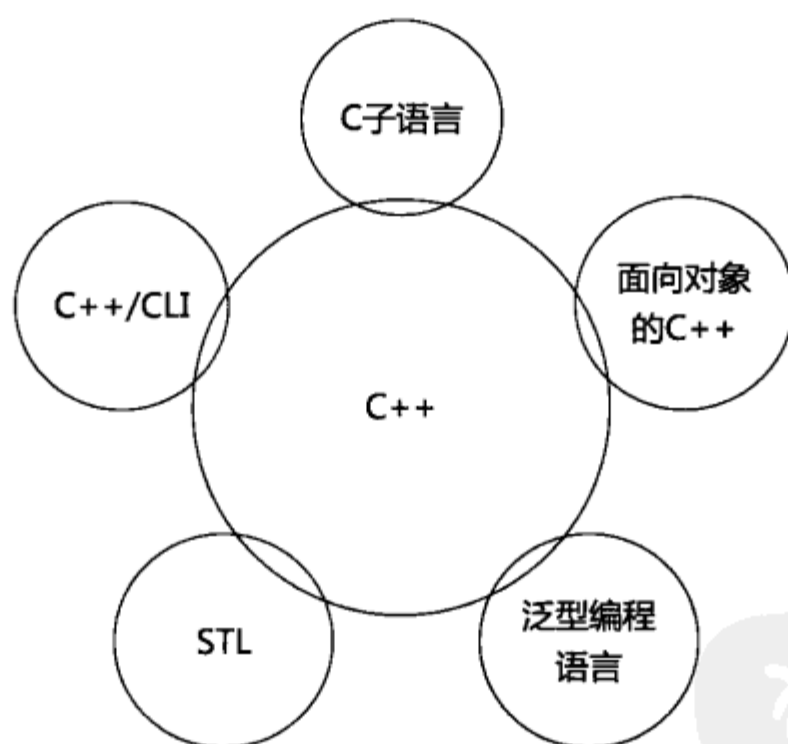


图 1-2 C++的五种争奇斗艳的子语言

现在的 C++世界，主要由以下五种子语言组成。

1. C 子语言

C++的发展渊源，使得 C++支持几乎全部的 C 语言功能，在语法上与 C 语言仅有细微的差别。很多人把 C++当做 C 语言来使用，形成了一种独特的子语言。

2. 面向对象的 C++

C++首先是作为一门面向对象的程序设计语言而闻名的。在应用中，C++也是一门优秀的面向对象的程序设计语言。

3. 泛型编程语言

泛型编程是独立于流行的面向对象编程的一种新的开发方式，可以编写完全一般化并可重复使用的算法，其效率与针对特定数据类型而设计的算法的效率相同。所谓泛型（genericity），是指在多种数据类型上皆可操作，与模板有些相似。C++强大的（但容易失控的）模板功能能够在编译期完成许多工作，使得它成为泛型编程的不二之选，C++也因此发展成一门独特的泛型编程语言。

4. STL

STL 是 C++泛型编程的一个杰出作品，随着 C++的不断发展，STL 变得越来越强大，它已经逐渐成为 C++程序设计中不可或缺的部分，其效率虽然比一般的 C++代码低，但是其安全性与规范性大受欢迎，在业界得到了广泛的应用，发展成为了一门独立于泛型编程之外的 C++子语言。

5. C++/CLI

微软为了让广大熟悉 C++的开发者能够在 .NET Framework 平台上进行应用开发，扩展 C++形成 C++/CLI，使得 C++/CLI 能借助强大的 .NET Framework 成为一门新的面向未来的 C++子语言。

1.3 C++世界版图

C++语言的发展过程，不仅是一个特性不断增加、内容不断丰富过程，更是一个在应用领域中不断攻城略地的过程。在其 40 余年的发展过程中，C++在多个应用领域都得到了广泛的应用和发展。无论是在最初的 UNIX/Linux 操作系统上，或者在 Windows 操作系统上，还是在最近兴起的嵌入式系统上，C++都占有一席之地，如图 1-3 所示。

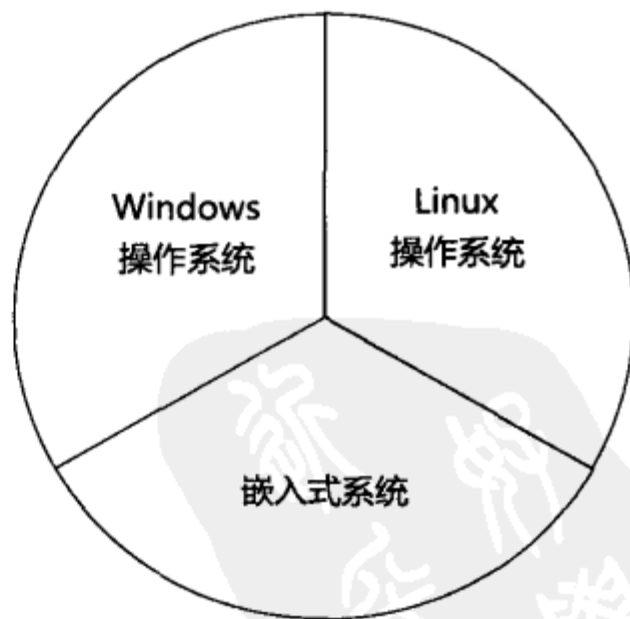


图 1-3 C++世界三分天下

1.3.1 Windows 系统下的 C++开发

自从“盖茨大叔”在 1985 年用 C++完成了 Windows 1.0 之后，C++就与 Windows 操作

系统结下了不解之缘。不仅 Windows 操作系统底层是使用 C++ 开发的，而且 Windows 操作系统上运行的大多数应用程序也是用 C++ 开发的，可以说 Windows 操作系统中流淌的是 C++ 的血液。虽然现在 Windows 操作系统上的程序设计语言有很多，但是 C++ 以其独特的优势在 Windows 平台上仍然占据不可撼动的地位。

1. 得天独厚的优势

Windows 操作系统本身是用 C++ 开发的，这使得 C++ 调用 Windows API 有着天然的优势，因此 C++ 是 Windows 操作系统上的首选程序设计语言。

2. 历史的积淀

由于历史的原因，有很多运行在 Windows 操作系统上的应用程序都是使用 C++ 开发的，这些应用程序需要继续维护和升级，因此 C++ 继续在 Windows 操作系统上保持先发优势。

3. 面向未来的 C++/CLI

因为微软的大力支持，C++ 获得 .NET Framework 的支持后摇身一变成为 C++/CLI，与 C# 等新兴程序设计语言并驾齐驱，共同开发 Windows 操作系统上面向未来的应用。

1.3.2 Linux 系统下的 C++ 开发

如果说在 Windows 操作系统下 C++ 还有其他的竞争者，那么在 Linux 系统下，C++ 几乎是开发者的不二选择了。

作为程序设计语言，C++ 在 Linux 操作系统下的优势非常明显。首先，Linux 操作系统本身是用 C 语言开发的，而 C++ 与 C 语言之间有着亲密关系，使得 Linux 操作系统本身对 C++ 开发非常友好。另外，Linux 操作系统上大多是服务器端的应用，这些应用强调高性能，这恰恰是 C++ 语言的优势。大多数 Linux 上的应用都是使用 C++ 开发的，比如著名的网络服务器 Apache、数据库服务器 MySQL 等。如果想在 Linux 操作系统上开发应用，那么 C++ 是首选编程语言。

1.3.3 嵌入式系统下的 C++ 开发

随着各种各样数码产品的流行，嵌入式系统成为了热门的开发领域。嵌入式系统特殊的硬件限制，使得嵌入式系统对开发语言有着特殊的要求。比如，嵌入式系统的内存容量比较小，要求对内存进行良好的管理；嵌入式系统的 CPU 主频比较低，要求可执行代码简洁高效；同时，为了提高开发效率，要求采用高级开发语言。嵌入式系统对开发语言的苛刻要求，正是 C++ 的优势所在。除了低级的汇编语言外，C++ 几乎是嵌入式系统开发的唯一选择。

1.4 如何学好 C++

既然 C++ 如此强大，那么如何学好 C++ 呢？

每个 C++ 初学者都会问这个问题。虽然这个问题没有统一的答案，但是作为一个 C++ 世界的导游，我可以介绍一些经验和教训给大家，让大家少走弯路，沿着正确的方向前进，轻松地愉快地完成 C++ 世界的奇妙之旅。

1.4.1 将自然语言转换为 C++ 程序设计语言

C++ 是一门程序设计语言，有着语言的基本特征，我们可以像学习普通语言一样来学习 C++。

语言，是用来描述和表达现实世界的，编程语言也不例外。为了描述现实世界的事物，我们需要一些名词。这些名词在 C++ 中就是数据类型和用数据类型表达的数据。为了表达事物之间的关系，可以将各个事物连缀成句子，这些句子在 C++ 中就是表达式。将多个句子通过一定的逻辑关系组合起来，就可以形成一篇文章。同样，在 C++ 中通过一定的逻辑控制将多个表达式组合起来就形成了程序。通过 C++ 编程语言和自然语言的对比，我们可以轻松地理解 C++ 程序的含义。C++ 是描述现实世界的编程语言，编写程序的过程，是将自然语言翻译成 C++ 语言的过程，如此而已。比如，在自然语言中，我们可以这样来描述一件事情：有个男孩叫小张，有个女孩叫小芳。男孩向女孩示爱。女孩对男孩进行考察，如果男孩有房又有车，则与男孩交往；如果没有，则与男孩拜拜。翻译过程可以参考图 1-4。

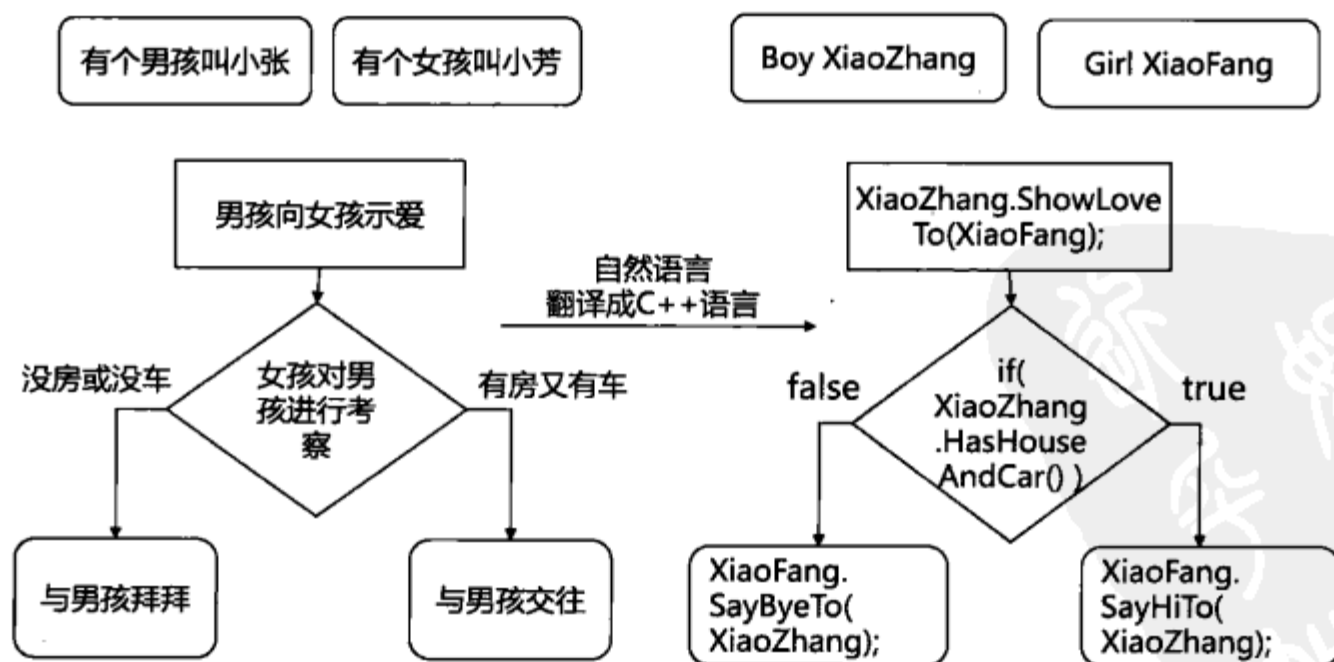


图 1-4 编程就是把自然语言翻译成 C++ 语言

如果把自然语言翻译成 C++ 语言，则是这样的：

```
//有个男孩叫小张
Boy XiaoZhang;
//有个女孩叫小芳
Girl XiaoFang;

//男孩向女孩示爱
XiaoZhang.ShowLoveTo(XiaoFang);
//女孩对男孩进行考察
//如果男孩有房又有车
if(XiaoZhang.HasHouseAndCar() )
{
    //女孩与男孩交往
    XiaoFang.SayHiTo(XiaoZhang);
}
else // 如果没有
{
    //则与男孩拜拜
    XiaoFang.SayByeTo(XiaoZhang);
}
```

通过将自然语言翻译成 C++ 编程语言，就形成了一段 C++ 程序。只要我们会说话，就会用 C++ 编程。C++ 编程，就这么简单！

虽然 C++ 语言是从 C 语言发展而来的，但是可以认为 C++ 是一门全新的独立的编程语言，它并不依赖于 C 语言。学习 C++ 不必掌握 C 语言，但学好了 C++ 语言，自然就掌握了 C 语言。

什么是程序设计语言

程序设计语言，通常简称为编程语言，它是一组用来定义计算机程序的语法规则，是一种标准化的交流技巧。利用程序设计语言，程序员能够准确地定义计算机需要使用的数据，并精确地定义在不同情况下所应当采取的行动，让计算机完成特定的任务。

程序设计语言是一套包含语法、词汇和含义的正式规范。这些规范通常分成 4 个部分。

- 数据成分：用以描述程序中所涉及的数据。
- 运算成分：用以描述程序中所包含的运算。
- 控制成分：用以表达程序中对运算流程的控制。
- 传输成分：用以表达程序中数据的传输。

按语言级别，程序设计语言可以分为低级语言和高级语言。低级语言包括字位码、机器语言和汇编语言。其特点是与特定的机器有关，效率高，但使用复杂、烦琐、费时、易出差错。其中，字位码是计算机可直接理解的唯一语言，但由于它是一连串的字位，复杂、烦琐、冗长，几乎无人直

接使用。机器语言是表示成数码形式的机器基本指令集，是操作码经过符号化的基本指令集。汇编语言把机器语言中的地址部分符号化，并进一步包括了宏构造。

高级语言是比低级语言更接近于待解决问题的表示方法，其特点是在一定程度上与具体机器无关，易学、易用、易维护。把高级语言程序翻译成低级语言程序，一个高级语言程序单位对应多条机器指令，产生的目标程序性能比低级语言程序低。C++语言就是一种高级程序设计语言。

大多数被广泛使用或经久不衰的程序设计语言，都有专门的标准化组织，负责规范及发布该语言的正式定义，并讨论扩展或贯彻现有的定义。

1.4.2 “多读多写”是学好 C++ 的不二法门

还记得当年我们学习外语的时候，老师总是教导我们要“多听多说多读多写”。同样，学习 C++ 也强调“听说读写”。对于编程语言而言，虽然没有“听”和“说”，但是“多读”和“多写”却同样是学好 C++ 的不二法门。

1. 多读

多读就是强调多阅读和学习别人的优秀代码，特别是一些优秀的开源产品的源代码。通过阅读这些源代码，不仅可以学习具体的语言知识、开发技术，还可以从中学习设计思想、编程风格等。向高手学习，是成为高手的唯一途径。

2. 多写

多写，就是多多地进行开发实践。编程是一门技艺，它来自于实践，光纸上谈兵是无法学好编程的。多写包括很多方面，比如，在开发环境中完成书本上的例子程序，重新实现网络上的例程，开发实现一些小程序等。多读的目的只是学习别人的知识和经验，多写的目的是将别人的知识和经验内化为自己的知识和经验。同时，通过多写可以发现很多在阅读技术书籍、阅读程序代码时隐藏的问题。通过自己编写程序、调试程序，可以获得宝贵的第一手开发经验，培养自己的动手能力，从而成为一个真正的高手。

1.4.3 和 Google 做朋友

在开发实践中，与其说程序是编出来的，倒不如说是搜出来的。如果遇到一个语法上的细节问题就可以用 Google 搜索关于 C++ 的教程；如果遇到函数使用上的问题，就可以用 Google 搜索这个函数的文档；如果遇到常见的开发任务，就可以用 Google 搜索已有的示例代码。甚至当遇到程序中的疑难杂症时，都可以用 Google 搜索有没有人遇到相同的问题。总之，编程开发离不开 Google。网络就像一个大金矿，而 Google 就是采矿机器。善用 Google、和 Google 做朋友，可以帮助我们充分利用丰富的网络资源来学好 C++，用好 C++。

你会使用 Google 吗？

这还用问吗？Google 谁不会用啊！

你真的会使用 Google 吗？

所谓搜索，就是“使用正确的工具和正确的方法寻找正确内容”。

这里的工具，就是我们使用的搜索引擎。Internet 只有一个，而搜索引擎则有许多个。虽然各个搜索引擎各有特长，但是对程序员来说，Google 是最合适的工具。至于原因，我就不在这里啰嗦了，大家可以自己去 Google。

掌握搜索引擎的使用技巧可以让我们轻松地在茫茫大海中捞出我们需要的那根针。例如，可以使用双引号来表示某个必须包含的关键词，也可以使用 OR 或者 AND 等逻辑运算连接多个关键词，从而更准确地表示我们想要的内容，甚至可以使用 site 标记，将搜索范围限定在某个网站之内。学习和掌握搜索引擎的使用技巧可以帮助我们更好地使用搜索引擎，更快、更准地找到我们想要的内容。



与 C++ 第一次亲密接触

在浏览了 C++ “三分天下”的世界版图之后，便对 C++ 有了基本的了解，算是跨入了 C++ 世界的大门。那么，下一步该往哪里去呢？即刻开始编写 C++ 程序？还是……

正在我们犹豫的时候，看到前面有一个人被一群满头问号的 C++ 初学者围在当中。我们赶紧挤进去一看，原来是一个 C++ 程序正做自我介绍呢。

2.1 一个 C++ 程序的自白

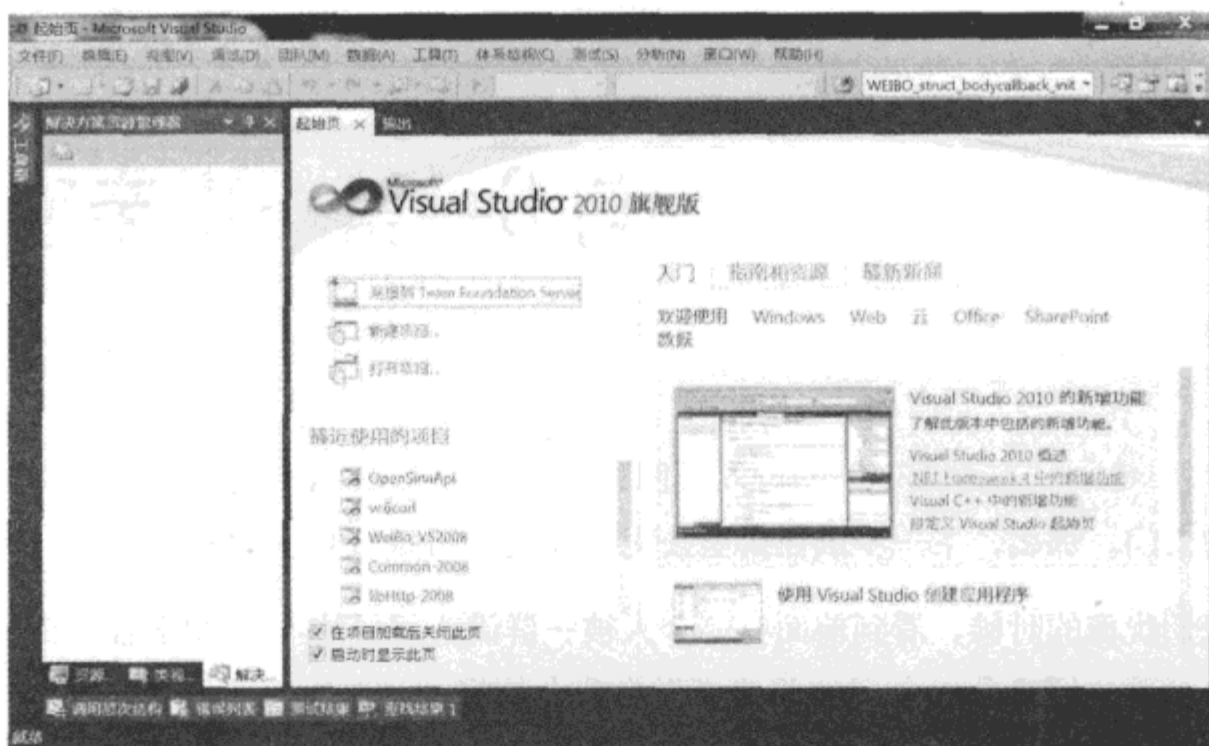
“大家好，欢迎来到奇妙的 C++ 世界。我是 C++ 世界的迎宾——一个最普通的 C++ 程序，我的名字叫“HelloWorld.exe”。我虽然很普通，但几乎是世界上最著名的 C++ 程序。每个来到 C++ 世界的初学者与 C++ 的第一次亲密接触都是通过我来完成的。大家在听我介绍之前，一定会觉得这些 C++ 程序非常神秘，比如，C++ 程序是如何创建的？完整的 C++ 程序由哪几部分构成？传说中的源文件是什么？C++ 程序是如何执行的？其实，我们一点都不神秘，跟大家一样，我们也同样是有血有肉的生命，我们有自己的老爸老妈，有自己的五官四肢，也有自己的生命过程。什么？大家觉得不可思议？别着急，下面且听我一一道来……”

2.1.1 用 Visual Studio 创建 C++ 程序

大家进入 C++ 世界最感兴趣的一件事，就是亲自动手创建一个 C++ 程序。大多数 C++ 程序都是通过一种叫集成开发环境（Integrated Development Environment, IDE）的软件创建的，可以说，它是创建 C++ 程序的工厂。虽然用于创建 C++ 程序的集成开发环境有很多，但是我们还是首选由微软公司开发的 Visual Studio。现在，请大家在我的引导下使用 Visual Studio 创建第一个 C++ 程序。

首先，启动已经安装好的华丽丽的 Visual Studio 啦！

待 Visual Studio 启动之后，弹出如图 2-1 所示的界面。大家可以通过以下 3 个步骤来创建 C++ 程序，也就是亲手创建一个 HelloWorld.exe。



1. 选择应用程序模板

从“文件”菜单中选择“新建→项目”，创建新的项目。Visual Studio 会引导我们进入模板选择界面。选择“Win32 控制台应用程序”模板，如图 2-2 所示。输入项目名称“HelloWorld”，然后选择项目存放的位置，单击“确定”后进入“Win32 控制台应用程序”的 AppWizard（应用程序向导）。大家可能会问，为什么要选择“Win32 控制台应用程序”呢？这是因为控制台应用程序没有与 Windows 用户界面相关的消息机制窗体控件等与 C++ 关系不大的复杂的技术，相对比较简单，对我们学习 C++ 语言完全够用了。



图 2-2 选择应用程序模板

在应用程序向导中，可以设置程序的具体参数，如图 2-3 所示。例如，选择应用程序的类型、是否需要 MFC 支持等。这里我们保持默认的设置就可以了，单击“完成”，Visual Studio 会创建一个基本的 Win32 控制台应用程序项目。

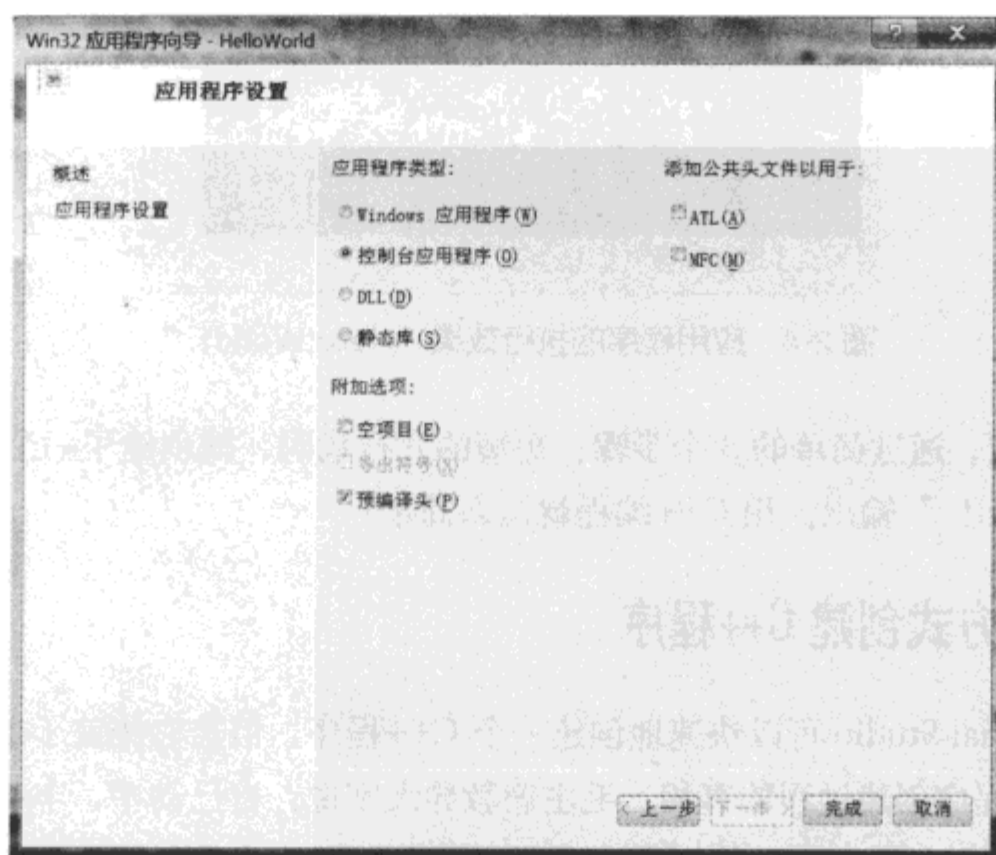


图 2-3 设置应用程序参数

2. 修改代码，实现功能

在第一步中，Visual Studio 创建了一个空应用程序，但它只是一个空的程序模板，并不具有任何功能。接下来要为其添加代码或者修改其中的代码以实现具体的功能。为了让这个程序可以向世界说“Hello World!”，我们在“解决方案资源管理器”中找到 HelloWorld.cpp 这个文件，双击打开，并做如下修改：

```
// HelloWorld.cpp : 定义控制台应用程序的入口点。
#include "stdafx.h"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    //在屏幕上显示“Hello World!”
    cout<<"Hello World!"<<endl;

    return 0;
}
```

3. 编译代码，构建应用程序

完成程序代码的修改后，就可以开始编译代码、构建应用程序了。从“生成”菜单中选

择“生成解决方案”，Visual Studio 就会编译生成 HelloWorld.exe 这个可执行程序。然后从“调试”菜单中选择“开始执行”，就可以看到程序的执行结果，如图 2-4 所示。

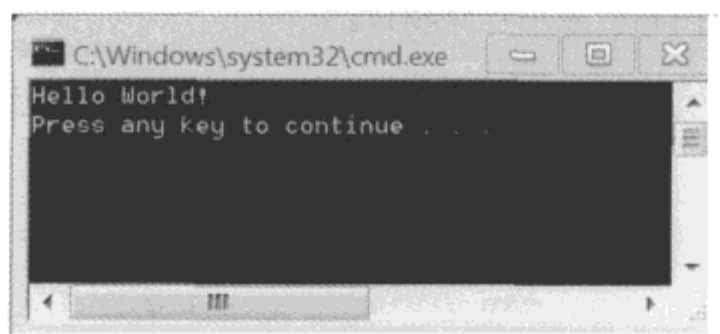


图 2-4 应用程序的执行效果“Hello World!”

大家可以看到，通过简单的 3 个步骤、短短的几行代码，就创建了一个 C++ 程序，并实现了“Hello World!”输出。用 C++ 编程就这么简单！

2.1.2 以手工方式创建 C++ 程序

虽然使用 Visual Studio 可以快速地创建一个 C++ 程序，但是它掩盖了很多细节，使人无法了解整个 C++ 程序创建过程的真相。毛主席教导大家说：自己动手，丰衣足食。为了了解 Visual Studio 创建 C++ 程序背后的故事，下面来看看如何以手工的方式创建一个 C++ 程序。

1. 创建源文件

手工方式创建 C++ 程序的第一步是直接创建应用程序源文件。可以使用记事本创建一个文本文件，命名为“HelloWorld.cpp”，其内容如下：

```
// HelloWorld.cpp : 以手工方式创建的 C++ 程序源文件
```

```
#include <iostream>
using namespace std;

int main()
{
    // 在屏幕上显示“Hello World!”
    cout<<"Hello World!"<<endl;

    return 0;
}
```

2. 编译源文件产生可执行程序

完成源文件的创建与编辑之后，接下来就将编辑好的源文件编译链接成可执行程序。可以在开始菜单的“Microsoft Visual Studio 2010”→“Visual Studio Tools”中找到“Visual Studio Command Prompt”。这是 Visual Studio 的 DOS 命令行窗口，可以方便地调用 Visual C++ 编译器编译源文件。首先在 DOS 命令行中将当前目录切换到源文件所在的目录，然后用下面的

命令编译链接源文件：

```
E:\MyFirstCPPBook\Source>cl /EHsc HelloWorld.cpp
```

其中，“E:\MyFirstCPPBook\Source”是源文件所在的目录，“cl”是编译链接命令，其后跟着的是编译选项“/EHsc”。Visual Studio 的 C++ 编译器有很多编译选项，用于指定编译器完成额外的功能，例如，可以通过“/OUT”选项指定输出文件的名称或者完整路径；通过“/Gm”选项启用编译器的“最小重新生成”功能，加快编译的速度；通过“/EHsc”启用 C++ 异常处理增加程序的健壮性；通过“/w”选项屏蔽编译过程中产生的警告信息等。执行“cl /?”命令，可以查看所有编译器选项的帮助信息。

在编译选项之后，就是要编译的源文件“HelloWorld.cpp”了。命令执行完成后，将在源文件目录下得到一个与源文件同名的可执行文件“HelloWorld.exe”。这样，这个可执行文件就在你手中诞生了。亲手创建一个 C++ 程序的感觉是不是很奇妙呢？上帝说，要有光，于是有了光；你说，要有 C++ 程序，于是有了 C++ 程序。

3. 运行 C++ 程序

创建完成 C++ 程序之后，应该想看看执行效果啦。很简单，只需要在命令行中输入程序的名字，回车执行就可以了：

```
E:\MyFirstCPPBook\Source>HelloWorld.exe  
Hello World!
```

这样，就可以在 DOS 窗口中看到我向世界发出的问候了。这一步相当于 Visual Studio 中的“开始执行”命令。与 Visual Studio 的 IDE 相比，命令行模式下的编译速度更快，对编译过程的控制也更加灵活。同时，命令行模式下的编译不会受 IDE 产生的附加信息的干扰。

2.1.3 C++ 程序=预编译指令+程序代码+注释

麻雀虽小，五脏俱全。大家别看我个头小，只有短短的几行代码，实现的功能也很简单，但是我同样拥有健全 C++ 程序的“五官和四肢”：预编译指令、程序代码和注释，如图 2-5 所示。大多数情况下，这三个基本部分都被放在一个扩展名为“cpp”的文本文件中，这个文件称为 C++ 源文件。源文件记录了我的“五官和四肢”，规划了我的人生。源文件的编写者就是我的设计师了。通过修改源文件，可以改变我的面貌、我的人生轨迹，让我完成各种任务。

下面大家一起来仔细看看我的源文件，从中认识我的五官和四肢。

1. 预编译指令

预编译指令以“#”开头，它们是发给编译器的命令，在编译源代码之前完成。在我的源文件中，有两个相似的预编译指令，具体如下：

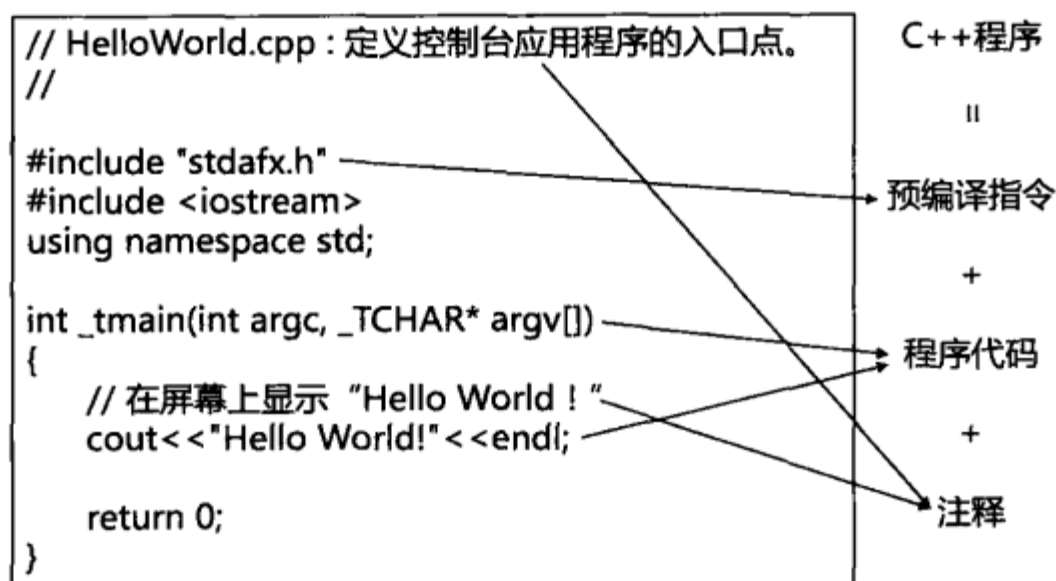


图 2-5 C++程序=预编译指令+程序代码+注释

```
#include "stdafx.h"
#include <iostream>
```

其中，“#include”指令用于在编译之前将指定的文件嵌入该指令所在的位置，作为整个源程序的一部分。这里嵌入了“stdafx.h”和“iostream”两个文件，这样就可以使用由这两个文件所定义的功能了。通常用这种方式调用 C++标准库的功能。注意，“#include”指令后的文件名有两种方式：如果使用“`""`”来表示一个文件名，则预处理器在处理这个指令的时候，将首先在当前目录下搜索这个文件，如果这个文件不存在，则继续在项目的包含目录下搜索这个文件；如果使用“`<>`”来表示一个文件名，预处理器则会直接在项目的包含目录下搜索这个文件。所以，通常使用“`""`”来嵌入当前项目目录下的文件，使用“`<>`”来嵌入各种项目包含目录下的库文件。

2. 程序代码

程序代码主体由若干 C++语句构成，可以说语句是程序的基本构成单位。在 C++中，每条语句用分号“`;`”结束。在我的源文件中，第一个 C++语句是：

```
using namespace std;
```

这条语句表示我所使用的名字空间是 `std`。所谓名字空间，就是标识符的上下文。同一个标识符可在多个命名空间中定义，它在不同命名空间中的含义是互不相干的。就像张家村有一个人叫陈良乔，而李家村也有一个人叫陈良乔一样，大家在称呼这个人的时候，必须明确地说明这个“陈良乔”到底是张家村的还是李家村的。所以，这里也必须说明要使用的是 `std` 名字空间下的各种标识符，这样编译器就知道程序代码中使用的标志符，例如 `cout`、`endl` 等，都是 `std` 名字空间下的标识符。

接下来就进入我的核心部分 `_tmain()` 函数了。这里 `_tmain()` 函数只有两条简单的语句。第一条语句是

```
cout<<"Hello World!"<<endl;
```

cout 是定义在头文件“iostream”中的一个输出流对象，它是 C++标准库预定义的对象，包含很多有用的输出功能。前面使用“#include”预编译指令包含“iostream”就是为了使用这个功能。关于输入/输出流，会在以后的章节中做更详细的介绍，这里只要知道这条语句可以输出字符串到屏幕上即可。

第二条语句是

```
return 0;
```

它表示程序成功执行完毕，返回一个值。到这里，_tmain()函数执行结束，我的生命也会在此终结。

3. 注释

注释是源代码的编写者为了提高程序的可读性写下的关于代码的一些说明，注释不会对程序的功能产生影响。在 C++中，可以使用“//”和“/* */”两种符号将一段文字表示为注释。

“//”是单行注释符，“//”之后直到换行的所有内容都属于注释。例如：

```
// 这是一行注释
```

“/*”和“*/”总是成对出现的，出现在这对符号之间的所有内容都属于注释。例如：

```
/*
```

```
这是一段注释
```

```
*/
```

从功能上讲，注释一般分为序言性注释和解释性注释。序言性注释多位于程序源文件的开始，用来说明程序的文件名、用途、编写时间、维护历史等。在上面的例子中，第一行注释是序言性注释，它说明了源文件的名称及功能，例如：

```
// HelloWorld.cpp: 定义控制台应用程序的入口点。
```

序言性注释被广泛用于大型的项目中。通常，每个项目都有自己定义的序言性注释格式，用来向阅读者说明一些必要的信息。下面是从一个实际的项目中摘录的一段序言性注释，它说明了源文件的名称、作用、创建时间、作者及其联系方式、文件的修改历史等信息，帮助阅读者理解代码。大家可以以此为模板，编写自己的序言性注释。

```
////////////////////////////////////  
// AppDataView.cpp : implementation file  
//  
// CAppDataView  
// This view is designed to display the App Data  
//  
// Version: 2.1  
// Date: September 2001
```

```
// Author: Chen Liangqiao
// Email: chenlq@live.com.
// Copyright (c) 2002. All Rights Reserved.
//
// History:
/*
27.09.2001      Chen Liangqiao.
                 Added OnCreate(), OnUpdate():
                 Added usage of mesh tracer layers
                 Added bugfix for Graphics zoom error
30.10.2001      Chen Liangqiao
                 Changed order of MPR View only in _TORCHTONAV
08.11.2001      Jia Wei
                 Added EUpdateReason, used for UpdateAllView(),
                 Added voxel trafo,
                 Changed the control panel due to new CTestCtrl
*/
```

////////////////////////////////////

与序言性注释不同，解释性注释多分散于程序的各个部分，用来向阅读者解释代码的含义，说明一些必要的问题等。例如，上面例子中的注释：

```
//在屏幕上显示 "Hello World!"
cout<<"Hello World!"<<endl;
```

这句解释性注释用来向阅读者说明其下代码的功能是输出字符串“Hello World!”。

最佳实践：什么是好的注释

虽然程序的注释并不影响程序功能的实现，编译器也不会去阅读我们的注释，但是好的注释可以增加程序的可读性，使程序易于维护。谁都不愿意维护一份没有注释的代码，那无异于阅读天书。那么，什么样的注释才是好注释呢？

注释是对代码的“提示和说明”，当认为代码难于理解的时候，或者需要特别说明的时候，就应该添加注释。但是，要防止注释过多。注释只是简短的说明性语句，不是文档。程序的注释不可喧宾夺主，注释太多会让人眼花缭乱。如果代码的含义很清楚，则不必加注释，否则多此一举，令人厌烦，例如：

```
i++;    // i 加 1
```

另外，应该养成良好的代码注释习惯。编写代码时添加必要的注释，修改代码时修改相应的注释，删除无用的注释，保证注释与代码的一致性。

注释应当准确、易懂，避免二义性。错误的注释不但无益而且有害。

注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。例如：

```
// 创建 MemoryBlock 对象
// 对下方的代码进行注释
MemoryBlock a(3);
MemoryBlock b(6);
```

```
swap(a,b); // 交换两个对象，对左侧的代码进行注释
```

如果代码比较长，特别是有多重嵌套时，应当在某些段落的结束处加注释，以便于阅读。例如，一个多重循环的代码及其注释如下：

```
for ( int i = 0; i < 100; ++i )
{
    for ( int j = 0; j < 100; ++j )
    {
        // 算法处理...
    } // for ( int j = 0; j < 100; ++j ) 循环结束
} // for ( int i = 0; i < 100; ++i ) 循环结束
```

虽然程序的注释并不影响程序的功能，但是可以增强代码的可读性。注释是 C++ 程序必不可少的部分。

2.1.4 编译器和链接器

虽然我是 Visual Studio 创建的，但是实际上，我的老爸和老妈是 Visual Studio 集成的编译器和链接器。此外，Visual Studio 提供的主要是编辑功能，可方便地编辑我的源代码。

我老爸编译器的工作是将高级语言 C++ 翻译为低级语言（机器语言）。

源文件是使用 C++ 这种高级程序设计语言编写的，以便于人们编写、阅读和维护。但计算机不理解高级语言，所以老爸的职责是将源程序翻译成计算机能够解读运行的目标语言（target language）。目标语言通常是汇编语言或目标机器的目标代码（object code），有时也称为机器代码（machine code）。通过老爸的工作，计算机能看懂 C++ 程序，就可以按照源文件中的指令执行相应的动作。

编译器完成程序的编译工作后，程序还只是一些目标文件，还需要链接器将一个或多个由编译器编译生成的目标文件和库函数链接成可执行文件，这样才能诞生一个可执行的 C++ 程序。下面再来回顾一下程序的诞生过程，如图 2-6 所示。

首先，我的设计师——你——在 Visual Studio 中用 C++ 语言编写我的源代码（source code），这些源代码是 .cpp 文件；然后，源代码文件经过预处理器（preprocessor）处理，执行源文件中的预编译指令，例如，展开源代码中的宏，引入相应的头文件等。接着，编译器开始工作，把预处理后的源文件编译成目标代码，形成目标文件。在 Windows 系统下，目标文件的扩展名一般是“obj”，而在 Linux 下目标文件的扩展名为“o”。

Visual Studio 集成的编译器是“cl.exe”，可以在 Visual Studio 的命令行窗口中手动执行 cl 命令，让编译源文件得到目标文件。例如，可以通过下面的命令编译 HelloWorld.cpp 源文件得到 HelloWorld.obj 目标文件。

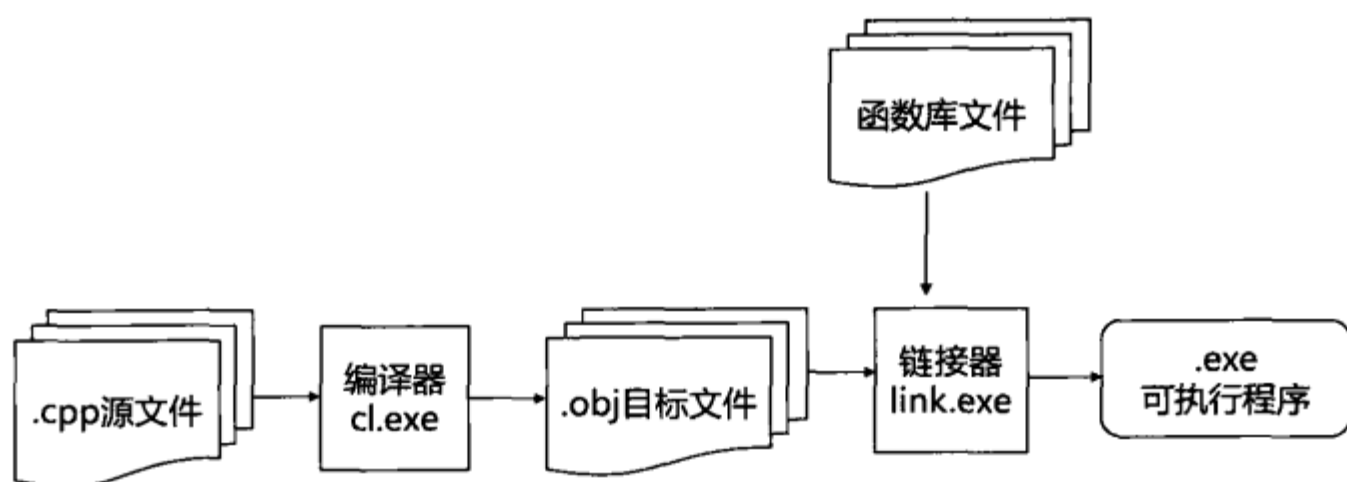


图 2-6 我的编译器和链接器

```
cl /c /EHsc HelloWorld.cpp
```

其中，cl 是编译源文件的指令，其后的参数（选项）用于指定编译器的行为。这里的“/c”表示只编译不连接；“/EHsc”指定编译器使用何种异常处理模型；最后一个参数 HelloWorld.cpp 是即将要编译的 C++ 源文件。

虽然目标文件中记录了计算机可以读懂的机器语言指令，但是这些目标文件还不能执行。C++ 程序通常由多个源文件组成，经过编译后形成多个目标文件，这些目标文件还需要最后组装成一个可执行程序。此外，程序一般会引用基本的库函数，因此在执行程序之前还要把库函数用定义好的目标代码替换。组装和替换目标文件的过程称为链接，也就是链接器的工作。从编译器那里得到编译好的目标文件后，链接器就忙碌起来了。链接器将编译器翻译的一个或多个目标代码文件跟库函数的目标文件进行链接，组合起来形成可执行程序。

在 Visual Studio 中，链接器是“link.exe”。在 Visual Studio 的命令行窗口中，你可以通过下面的命令让链接器链接目标文件 HelloWorld.obj，得到可执行的 HelloWorld.exe 文件：

```
link HelloWorld.obj
```

恭喜你亲手创建了一个 HelloWorld.exe。

2.1.5 C++ 程序的执行过程

一旦生成可执行文件，就可以给操作系统下达指令让文件运行。操作系统接收到命令之后，首先要创建相应的进程并分配私有的进程空间，加载器负责把可执行文件的数据段和代码段映射到进程的虚拟内存空间中，并加载所链接到的库函数。然后，操作系统开始初始化所定义的全局变量，自动调用全局对象的构造函数。最后，才进入入口点函数，也就是 _tmain() 函数开始执行。

进入 _tmain() 函数后，程序会按照源代码制定的人生规划，一条语句一条语句地往下执行，一步一步地往下走。你一定还记得，我的源代码是这样的：


```
int _tmain(int argc, _TCHAR* argv[])
{
    // 在屏幕上显示 "Hello World!"
    cout<<"Hello World!"<<endl;

    return 0;
}
```

_tmain()函数有两个参数 argc 和 argv,可以通过这两个参数给_tmain()函数传递一些信息,给我一些额外的吩咐,比如通过这两个参数告诉我应当在屏幕上显示什么内容等。在这里,暂时没有使用这两个参数。

进入_tmain()函数后,我遇到的第一条语句就是:

```
cout<<"Hello World!"<<endl;
```

这条语句让我在 DOS 窗口中显示 "Hello World!" 这样一个字符串,于是开始控制 DOS 窗口,在其中显示这个字符串,完成人们交给我的任务。

接下来的一条语句是:

```
return 0;
```

这条简短的语句宣告了我人生历程的结束。它表示整个_tmain()函数的结束。图 2-7 所示的是我短暂而光辉的一生!

```
int _tmain(int argc, _TCHAR* argv[]) ←—— 我的生命从这里开始
{
    { // 在屏幕上显示 "Hello World!" } ←—— 这是我一生要完成的事业
    { cout<<"Hello World!"<<endl; }
    return 0; ←—— 我的生命在这里结束
}
```

图 2-7 Hello World 程序短暂而辉煌的一生

2.1.6 程序的两大任务：描述数据与处理数据

每个人都会问自己人生的目的是什么？我的人生目的是什么？人们编写程序的目的是用程序解决现实世界中的问题。人们观察发现，这些问题都是以数据作为输入，然后对这些数据进行处理，最后得到问题的结论的。所以，我人生的目的是描述数据并处理数据，最终解决现实世界的问题，如图 2-8 所示。

人们用公式给我下了一个定义：

程序 = 数据 + 算法

其中数据可以看成是对现实世界中的各个事物的抽象。例如：在数学中用到的整数、小数，

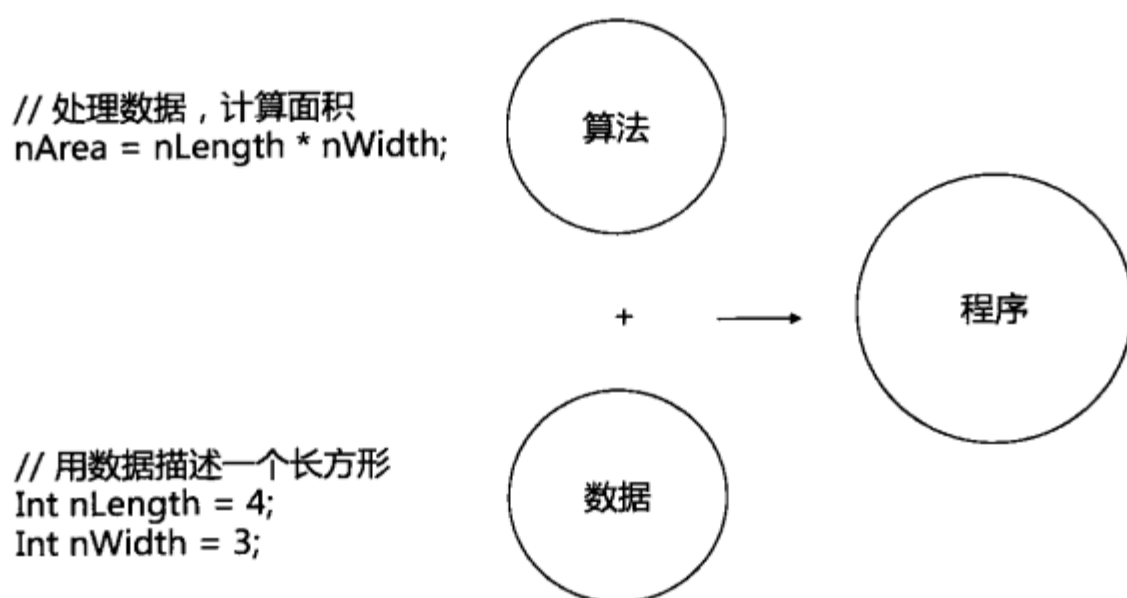


图 2-8 我的人生目的

我用数值数据来表示；在生活中用到的地名、人名等，我用字符串数据来表示。当然，还有其他的数据，我都有相应的表示方式。用数据描述现实世界中的事物，是我的人生目的之一。

现实世界是由这些事物组成的，同时各个事物又是变化的，各个事物之间还有各种关系。如果说数据只是现实世界的静态反映，那么对数据的处理就反映了事物的变化，表达了事物之间的关系，让整个世界运转起来。对数据处理的抽象，人们称为算法。通过设计算法，我可以帮助人们解决很多现实生活中的问题，实现我的另一个人生目的。例如，我可以处理学生成绩，得到平均成绩；我也可以处理长方形的长和宽，得到这个长方形的面积。

数据和算法伴随我的一生。在小小的 HelloWorld.exe 中，也同样有数据和算法的存在。例如，向屏幕输出“Hello World!”的语句：

```
cout<<"Hello World!"<<endl;
```

其中“Hello World!”是要向屏幕输出的数据，它是一个字符串数据。整个语句则代表了对这个字符串数据的处理：将字符串显示到屏幕上。数据和算法总是这样形影不离，成为我一生的目的。

2.2 基本输入/输出流

听过 HelloWorld.exe 的自我介绍之后，大家已经知道了一个 C++ 程序的人生目的就是描述数据和处理数据。现在的问题是，数据不能无中生有，C++ 程序也不能凭空创造出来数据。那么，C++ 程序中的数据又从何而来呢？

在现实世界中，国与国之间的交流是通过外交官来完成的。在 C++ 世界中，也有负责应用程序跟外界进行数据交流的外交官，它们的名字就是基本输入/输出流（iostream）。C++

语言使用标准类库 (standard library) 中的 `iostream` 类库实现基本的数据输入/输出。一个 C++ 程序在工作的时候, 负责输入 (`istream`) 的外交官专门负责将现实世界中的数据输入程序中, 然后 C++ 程序才能对这些数据进行处理。当 C++ 程序得到计算结果之后, 负责输出 (`ostream`) 的外交官又会将计算结果输出到屏幕或者以文件形式供大家查看。这两位外交官通力合作, 完成 C++ 程序与外界的数据交流。

实际上, 在前面的例子中用到的 `cout` 就是由 `iostream` 类库提供的。`iostream` 由两个基本的子类型 `istream` 和 `ostream` 构成, 分别表示输入流和输出流。C++ 程序将数据从一个对象到另外一个对象的流动称为流 (stream)。这样的定义比较抽象, 可以这样理解: 将程序中的数据显示到屏幕上, 或者写入文件中, 这个信息从程序流动到外部的过程是输出; 反过来, 数据从外部流动到程序的过程就是输入, 比如数据从键盘输入流动到程序内部数据, 或者从外部文件流动到程序内部。下面就来看看 C++ 世界的外交官是如何完成基本输入/输出的。

2.2.1 标准的输入和输出对象

`iostream` 类库定义了 4 个最基本的输入/输出 (I/O) 对象, 其中最常用的是 `cin` 对象和 `cout` 对象。`cin` 对象用来处理标准输入, 即键盘输入; `cout` 用来处理标准输出, 即屏幕输出。另外, 类库还定义了两个输出对象, 分别是 `cerr` 和 `clog`。`cerr` 对象用来处理标准的错误, 典型情况下用于生成警告或错误消息; `clog` 对象用于生成程序的执行信息。

有了输入对象和输出对象, 就可以通过预定义的插入符 “<<” 和提取符 “>>” 向流中插入数据或者提取数据。例如, 可以使用 “<<” 插入符向 `cout` 对象中插入数字或者字符串, 将其显示到屏幕上:

```
cout<<1;                // 向输出对象中插入数值 1
cout<<"Hello World!";    // 向输出对象中插入字符串 "Hello World!"
cout<<"1 + 2 = "<<1+2;    // 向输出对象中插入字符串 "1 + 2 =" 以及 1+2 的计算结果
```

第一句中的插入符将数字 “1” 插入输出对象中, 这样就会在屏幕上显示数字 1。同理, 第二句会在屏幕显示一个字符串 “Hello World! ”。最后一条语句中, 第一个插入符首先将 “1+2=” 这个字符串数据插入输出对象, 然后计算 “1+2” 的值, 最后将计算结果 “3” 插入输出对象, 最终显示在屏幕上的就是 “1 + 2 = 3”。

对于输入对象 `cin`, 可以使用提取符 “>>” 从 `cin` 输入流中获取用户输入的数据。例如:

```
// 用于保存用户输入数据的变量
int nAge;
string strName;
// 从输入对象 cin 中提取用户输入的整数数据和字符串数据,
// 分别保存到 nAge 和 strName 变量中
cin>>nAge>>strName;
```

在这里，首先定义了两个变量，分别用于保存用户输入的整数数据和字符串数据。然后，利用提取符“>>”从输入对象 cin 中提取用户输入的数据并分别保存到相应的变量中，这样就完成了数据从键盘到应用程序的输入。

下面再来看一个输入和输出配合使用的实例。

```
#include "stdafx.h"
// 引入定义输入/输出流对象的头文件
#include <iostream>
// 引入 std 名字空间
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 在屏幕上输出提示信息
    cout<< "请输入两个整数: " <<endl;
    int v1, v2;
    // 从输入对象提取用户输入的整数
    cin>> v1 >> v2;
    // 将计算结果输出到屏幕
    cout<< "两个整数" << v1 << "和" << v2
        << "的和是" << v1 + v2 <<endl;

    return 0;
}
```

通过简单的输入/输出，该程序实现了加法计算功能。下面来分析一下这个程序，看看它是如何实现数据的输入和输出的。程序的第 1 行和第 2 行是预编译指令，其中第 1 行是由 IDE 自动产生的，不必在意。这里要详细介绍的是第 2 行：

```
#include <iostream>
```

这条预编译指令把 iostream 文件引入当前文件中，因为需要用到的 cout 和 cin 对象定义在这个文件中。只有引入这个文件，才能在程序中使用 cout 和 cin 对象。当然，还需要使用 std 这个名字空间，否则当编译器找不到相应的标识符时会产生编译错误。

接下来从主函数_tmain()开始，分析程序的执行过程。

```
cout<< "请输入两个整数: " <<endl;
```

这条语句首先将字符串“请输入两个整数：”插入输出对象 cout 中，然后插入一个特殊值 endl，这个特殊值称为操纵符，稍后将介绍更多的操纵符。endl 操纵符的作用就是输出换行符，并且刷新输出缓冲器，确保用户立即看到已经插入输出流的信息。

显示提示信息后，下一步需要做的是接收用户的输入。为了保存用户输入的数据，先定义两个整型变量 v1 和 v2：

```
int v1, v2;
```

然后，开始使用输入流对象 cin 来获取用户的输入：

```
cin>> v1 >> v2;
```

程序运行到这里会停止，等待用户输入。输入两个数，然后回车表示输入完成。读取符“>>”会从输入流中读取这两个数，然后分别保存到 v1 和 v2 中。

获取数据之后，就可以开始运算并向用户输出运算结果了：

```
cout<< "两个整数" << v1 << "和" << v2
      << "的和是" << v1 + v2 <<endl;
```

这条语句将提示字符串和变量 v1、v2 的值，以及加法运算表达式的计算结果值插入输出流，最终在屏幕上显示计算结果，完成数据的输出。

2.2.2 输出格式控制

输出数据时，对不同的数据类型往往有不同的格式要求，比如小数的精度、输出数字的宽度等，需要对输出流的格式进行控制以满足要求。

为了控制输出流格式，C++提供了很多操纵符。这些操纵符可以直接插入输出流中以控制输出格式，它们都定义在头文件 iomanip 中，使用这些操纵符，需要先使用预编译指令 #include 引入这个头文件。表 2-1 列出了 C++中常用的格式操纵符。

表 2-1 常用的输出流格式操纵符

操 纵 符	作 用
dec	采用十进制表示数值数据
hex	采用十六进制表示数值数据
oct	采用八进制表示数值数据
endl	插入换行符，并刷新流
setprecision(int)	设置浮点数的精度，精度是浮点数中所有包括小数点前后的十进制数字的个数
setw(int)	设置输出流中两个数据显示的间隔宽度
setiosflags()	输出流的默认对齐方式为文本右对齐，程序中可以用 setiosflags 和 resetiosflags 操纵符重设对齐方式

例如，要求显示浮点数 1.23456 小数点后两位有效数字，然后换行，可以用如下的语句：

```
cout<<fixed<<setprecision(2)<<1.23456<<endl;
```

首先向 cout 对象插入一个 fixed 操纵符，让它以普通的小数计数法输出，否则它将以科学计数法输出浮点数；然后，通过 setprecision()设置需要保留的有效数字位数，这样就可以达到对输出格式的要求了。

有时候还需要对字符串的格式进行控制，从而让程序的输出更加美观。可以在字符串中加入一些用于格式控制的转义字符。常用的格式控制转义字符有：“\n”表示换行；“\t”表示间隔一个 Tab 的距离等。例如，下面的代码实现了换行显示：

```
cout<<"分多行\n 显示一个字符串"<<endl;
```

程序执行后，将在屏幕上看到“\n”将一个字符串分成了两行显示：

```
分多行
显示一个字符串
```

综合使用 C++ 语言所提供的这些输出流操纵符和格式定义转义字符，可以实现丰富的自定义格式化输出，满足对输出格式的各种要求。

2.2.3 读/写文件

除了用设备进行输入/输出之外，更多的时候，程序需要跟文件进行数据的输入/输出，这时候就需要对文件进行读/写操作：从文件读取数据到应用程序进行处理；将数据写入文件进行保存。C++ 提供了两个类 `ofstream` 和 `ifstream` 以实现应用程序跟文件之间的数据交流。

`ofstream` 和 `ifstream` 是由 `<fstream>` 头文件定义的两个类，分别负责数据的输出和输入。可以分别创建这两个类的对象，通过这些对象提供的函数打开文件，从而将对象跟某个具体的文件联系起来，再利用插入符“<<”和提取符“>>”对这些对象进行操作。例如，下面这段程序使用 `ofstream` 和 `ifstream` 两个类实现文件的读/写：

```
#include "stdafx.h"
#include <iostream>
// 引入输入/输出文件流对象需要的头文件
#include <fstream>

using namespace std;
// 主函数
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义变量，保存程序中的数据
    int nYear, nMonth, nDate;
    // 尝试打开 Date.txt 文件，并将其连接到输入文件流 fin
    ifstream fin("Date.txt");
    // 如果成功打开 Date.txt 文件，则从文件中读取内容
    if( !fin.bad() )
    {
        // 忽略文件中第 1 行的提示信息
        fin.ignore(256, '\n');
        // 用提取符 ">>" 从文件输入流 fin 中读取记录的数据，并保存到相应的变量
        fin>>nYear>>nMonth>>nDate;
        // 将数据显示到屏幕
        cout<<"文件中的日期是"
```

```

        <<nYear<<"-"<<nMonth<<"-"<<nDate<<endl;
    // 读取完成后, 关闭文件
    fin.close();
}
else
{
    // 如果文件打开失败, 则提示错误信息
    cout<<"无法打开文件并进行读取"<<endl;
}

// 提示用户输入新的数据并将其写入文件
cout<<"请输入当前日期(年月日): "<<endl;
// 从用户屏幕获取用户的键盘输入并保存到相应的变量中
cin>>nYear>>nMonth>>nDate;

// 尝试打开文件 Date.txt, 并将其连接到输出文件流 fout 中
ofstream fout("Date.txt");
// 如果成功打开 Date.txt 文件, 则将用户输入的数据写入文件
if( !fout.bad() )
{
    // 利用插入符 "<<" 将数据写入文件输出流 fout 中,
    // 也就是将数据写入文件中
    fout<<"用户输入的当前日期是: \n"
        <<nYear<<" "<<nMonth<<" "<<nDate;
    // 写入完成后, 关闭文件
    fout.close();
}
else
{
    // 如果无法打开文件, 则提示用户信息
    cout<<"无法打开文件并进行写入"<<endl;
}

return 0;
}

```

在这段程序中, 首先创建了一个输入文件流 `ifstream` 的对象 `fin`, 并利用它的构造函数将其连接到一个文本文件 `Date.txt`。所谓构造函数, 就是这个对象创建时所执行的函数。这里, 使用“`Date.txt`”作为参数来调用这个构造函数, 实际上就是使用这个文件创建 `fin` 对象。除此之外, 还可以使用 `fin` 所提供的 `open()` 函数来打开一个文件。当利用 `fin` 成功打开一个文件之后, 就可以利用提取符“`>>`”从 `fin` 中提取各种数据。“`>>`”会以空格为分隔符逐个从文件中读取数据, 并将其保存到相应的数据变量中。例如, 如果文件中的内容如下:

```

用户输入的当前日期是:
1983 7 3

```

默认情况下, `fin` 总是从文件的开始部分进行读取的, 为了直接读取第 2 行的内容, 可

以使用“`fin.ignore(256, '\n');`”忽略第 1 行的内容，将读取位置跳转到第 2 行。然后，通过提取符“`>>`”将第 2 行用空格分割的三个数据分别提取并保存到三个变量中。同样，为了将数据写入文件，需要创建一个输出文件流 `ofstream` 的对象 `fout`，然后通过它的构造函数或 `open()` 函数来打开一个文件，将这个文件和 `fout` 对象连接起来，然后通过插入符“`<<`”将数据插入 `fout` 对象，这样就实现了将数据写入它所关联的文件中的目的。整个过程如图 2-9 所示。

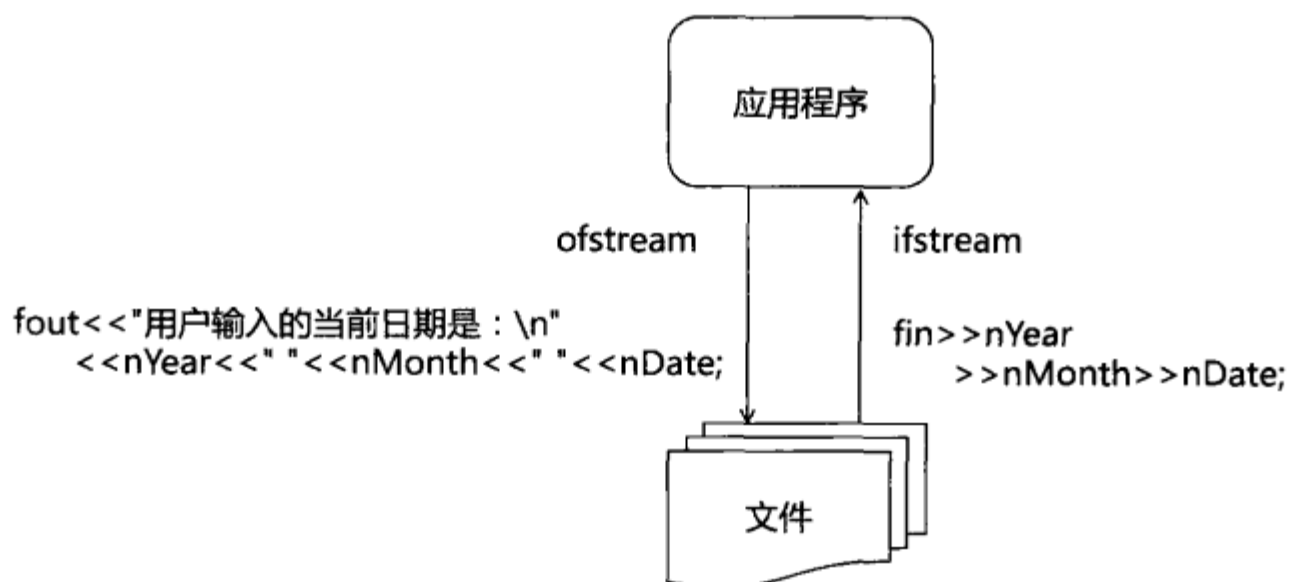


图 2-9 文件读/写

除了上面所介绍的文本文件的读/写之外，C++所提供的 `fstream` 还可以读/写二进制文件，甚至可以重新定义“`<<`”和“`>>`”，实现特殊的读/写操作。

2.3 最常用的开发环境 Visual Studio

刚刚听完了 `HelloWorld.exe` 的自我介绍，又看到几个 C++ 集成开发环境在一起聊得欢：Eclipse 说它使用起来很简单，Dev-C++ 笑了；Dev-C++ 说它开发效率高，C++ Builder 笑了；C++ Builder 说它资格老，Turbo C 笑了；Turbo C 说它粉丝多，Visual C++ 笑了……

知道更多：C++ 集成开发环境

集成开发环境（Integrated Development Environment，IDE）是用于开发程序的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面工具等，是集成代码编写功能、分析功能、编译功能、调试功能的开发软件套装。新的 IDE 甚至融合了建模功能、测试功能、项目管理等，完整覆盖了应用程序的整个生命周期。

除了 Visual Studio 之外，能够用于 C++ 开发的 IDE 还有很多，主要包括：

- Eclipse CDT
- Turbo C++
- C++ Builder
- Dev-C++
- Visual C++

在 Windows 平台上，应用最广泛的还是 Visual C++，很多现在流行的应用程序都是使用它开发出来的。同时，经过多年的不断改进，Visual C++已经成为了最为成熟的 C++集成开发环境之一，成为了程序员们心目中的首席 IDE。但是，就学习 C++而言，Dev-C++则显得更加小巧和灵活，同样是一个不错的选择。

微软公司的 Visual C++通常又简称为 VC 或 MSVC，是其开发套件 Visual Studio 的一个重要组成部分。微软自 1992 年推出 Visual C++ 1.0 以来，经过 10 多年不断的发展，Visual C++已经成为了 Window 平台下功能最强大、应用最为广泛的可视化应用程序开发工具之一。在某些领域，比如操作系统编程、游戏开发、图形图像处理、COM 编程、网络编程等，Visual C++具有不可比拟的优势，成为很多程序员的首选开发工具。

Visual C++自发布以来，版本不断更迭，现在最新的版本是集成在 Visual Studio 2010 中的 Visual C++ 2010。借助 WPF 打造的全新 IDE，Visual C++ 2010 带来了前所未有的编程体验，让我们体验到了编程的快乐。

工欲善其事，必先利其器。虽然可以以手工的方式创建 C++应用程序，但是在清楚 C++应用程序创建过程的前提下，合理使用 IDE 来创建 C++应用程序，无疑会大大提高效率。为了能够更好地进行 C++编程开发，下面先来介绍 Visual Studio 这位首席 IDE。

2.3.1 Visual C++的常用菜单

一个饭店提供的饭菜是否丰富，只要看它的菜单就知道了。Visual C++提供了长长的菜单，准备了丰盛的大餐，下面来看看 Visual C++准备了哪些好菜吧！

1. 文件菜单

文件菜单是 Visual C++中最常用到的菜单项，创建新工程、新文件，以及保存、关闭工程都要用到它。我们总是使用文件菜单来打开或者关闭某些项目，所以，就当这个子菜单是饭前的小菜或者饭后的甜点吧。

展开后的完整文件菜单如图 2-10 所示。整个菜单被分隔线分成了 8 组，相近的功能被划分在一组，在这个菜单中，最常用的有以下几项。

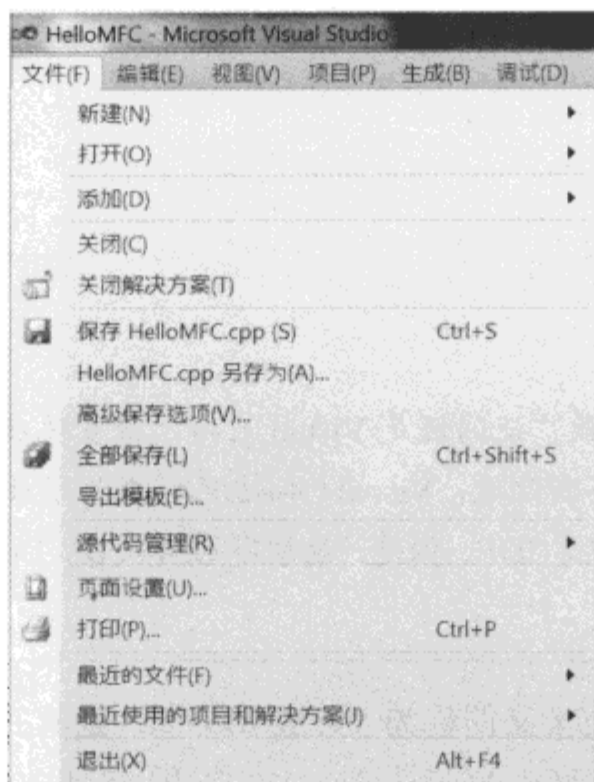


图 2-10 文件菜单

- 新建：用来创建新的工程、文件、网站等。以创建新文件为例，单击它的下级菜单“文件”后，出现如图 2-11 所示的文件类型选择对话框。

从图 2-11 所示的对话框中可以选择需要创建的文件类型，单击“打开”按钮，Visual C++ 会创建相关的文件。Visual C++ 支持的文件主要有 C++ 源文件、C++ 头文件、位图文件、图标文件、文本文件等。



图 2-11 选择新文件的类型

- 打开：用来打开已经存在的工程、文件、网站，以及进行工程的转换。
- 添加：将新的或者已经存在的工程、网站等添加到当前解决方案中。

- 关闭解决方案：关闭当前打开的解决方案。
- 最近使用的项目和解决方案：用来快速访问最近使用过的工程。

知道更多：菜单项后的省略号

在某些菜单项的后面有省略号，例如，新建文件的子菜单项“File...”。这里的省略号表示单击该菜单项后会弹出一个对话框，让用户进行进一步的操作，就像上面提到的弹出一个对话框供用户选择文件类型。这是 Windows 应用程序的一个设计惯例，以后在同样的情况下，也应当采用这样的设计，带有对话框的菜单项应该加上省略号。

2. 编辑菜单

一个优秀的 IDE，首先是一个优秀的代码编辑器。Visual C++ 是一个功能强大的代码编辑器，提供了很多有用的功能来帮助我们高效地编辑源代码，如图 2-12 所示。编辑菜单中常用的功能有以下几项。

- 撤销：撤销上一次的编辑动作。
- 重做：恢复撤销的动作。
- 剪切：将选中的内容剪切到剪贴板中。
- 复制：将选中的内容复制到剪贴板中。
- 粘贴：将剪贴板中的内容粘贴到当前位置。

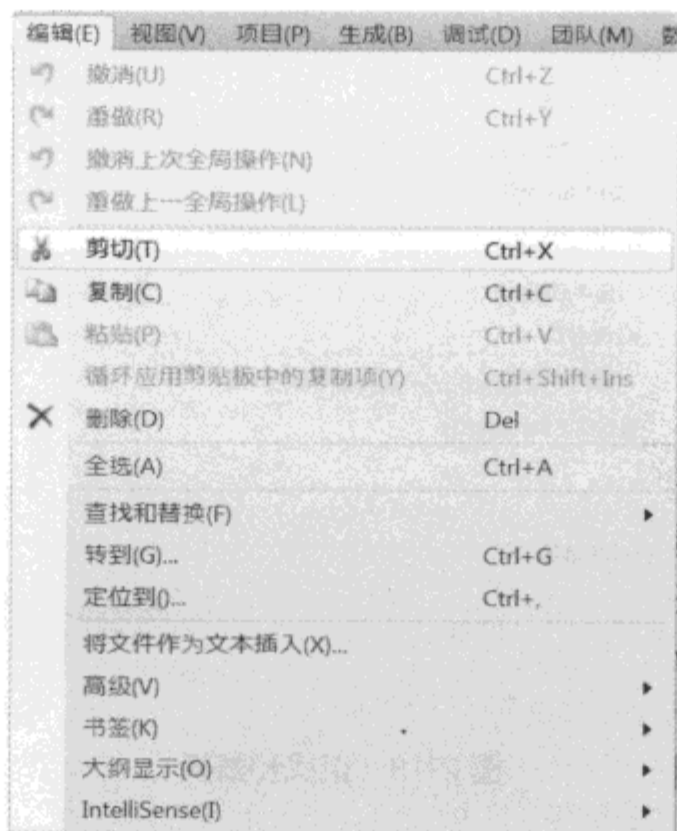


图 2-12 编辑菜单

- 循环应用剪贴板中的复制项：这是 Visual C++ 的特色功能，可以将剪贴板中的内容循环粘贴出来，直到得到想要的内容。这样就扩展了剪贴板的容量，可以保存多次复制到剪贴板中的内容。
- 全选：选中当前编辑窗口中的全部内容。
- 查找和替换：在单个文件或者多个文件中查找替换字符串。这是编辑菜单中非常有用的功能，特别当项目较大时，合理地使用这个功能，可以快速地找到需要的内容。下面来看看“查找和替换”功能具体的使用方法。

首先，单击它的下级子菜单“在文件中查找”，Visual C++ 会弹出如图 2-13 所示的查找对话框。可以在其中输入要查找的内容、查找范围等，从而精准地找到需要的内容。

- (1) 输入想要查询的字符串，通常是变量或者函数的名字。
- (2) 选择想要搜索的范围，这里已经设定了一些常用的路径，也可以根据需要进行特殊的目录。
- (3) 在“查找选项”中，可以对搜索做一些更细致的设置：“大小写匹配”表示在搜索的过程中区分大小写；“全字匹配”表示匹配整个单词；“使用”表示选择使用正则表达式还是使用通配符来完成更加复杂的查找，默认情况下使用正则表达式。
- (4) 在“结果选项”中，可以设置结果的输出位置，让搜索结果显示在需要的位置。

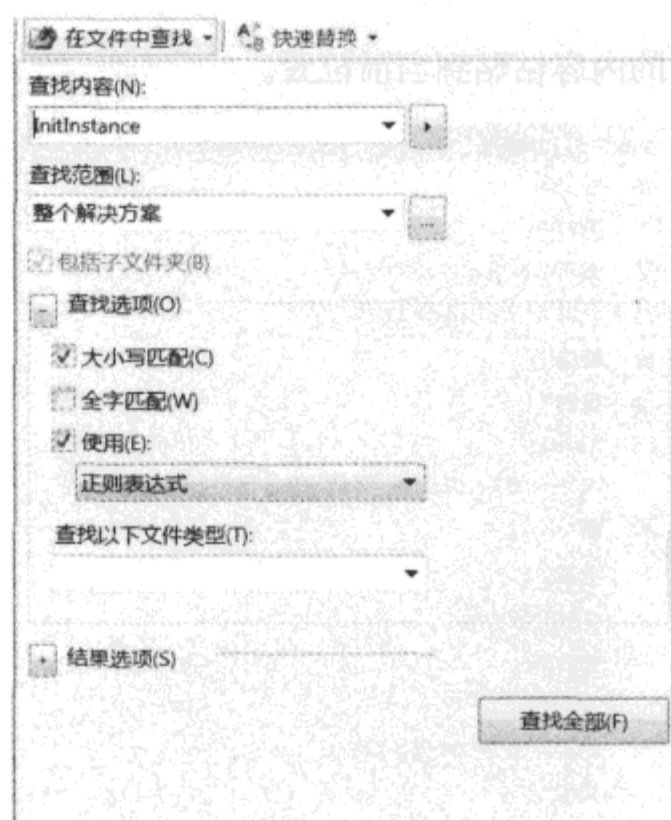


图 2-13 查找和替换

- 高级：这个菜单项包含了很多高级的编辑手段，例如，整理选中的代码格式，将选中的代码注释掉、变为大写，修改代码的缩进等。

知道更多：让你的代码立正

干净整洁的环境谁都喜欢，同样，格式整齐的代码谁都爱看。但是，反复修改可能破坏原来的代码格式。这时，高级编辑功能中的代码格式化书写就可以大派用场了，它可以将凌乱的代码按照语法规则调整格式，让代码立正，变整齐，增加了代码的可读性。

这个功能的快捷键是 Ctrl+K、Ctrl+F。

时刻牢记，让代码保持整齐，是一个良好的习惯。

3. 项目菜单

Visual C++的项目菜单如图 2-14 所示，该菜单主要用来向项目中添加一些新的类和资源等，同时对项目的属性进行具体的设置。下面介绍项目菜单中常用的几个功能。

- 添加类：向项目中添加新的类。在实际开发过程中，常常要向项目添加新的类来扩展其功能，这个菜单项，就是用来方便创建新类并将其添加到当前项目中。单击这个菜单，会出现如图 2-15 所示的添加类的对话框，用于选择要创建的类的模板。Visual C++已经准备了很多模板，帮我们省去了很多烦琐的工作。下面来看看如何向项目中添加新类。单击“添加类”菜单项，在弹出的类模板选择对话框中选择类的模板，这里选择 Visual C++下最常用的 C++类，然后单击“添加”按钮进入下一步，设置类的相关参数。

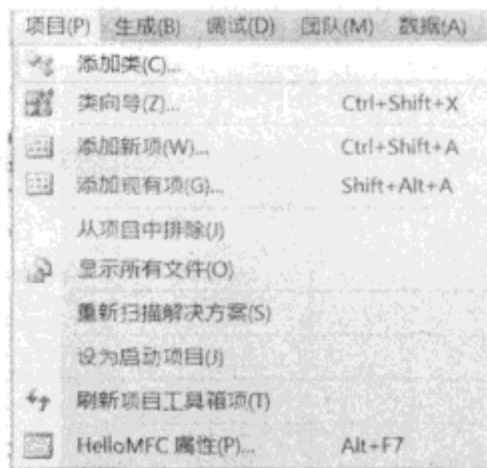


图 2-14 项目菜单

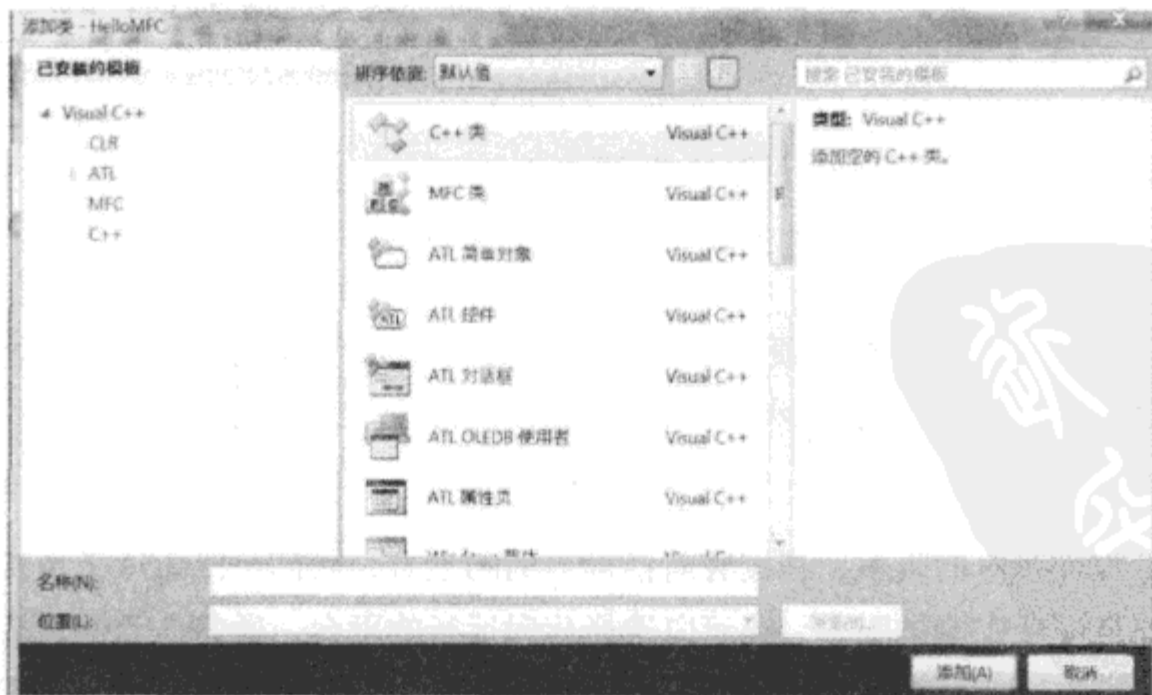


图 2-15 选择添加的类模板

单击“添加”按钮后，Visual C++会提供一个类的属性设置对话框，用于输入要添加新类的相关信息，这里可以输入类的名字、基类的名字及这个类所在的文件等。设置完成后，Visual C++将会按照其中的设置创建一个空的新类，包括相应的头文件和源文件。

- 添加新项：通过这个菜单项可以向项目中添加新的资源、UI、代码，甚至实用工具等，但是更常用的还是利用它向项目中添加资源。选择添加资源后，会出现一个对话框以选择资源类型。Visual C++的常用资源类型基本上都包括在其中了，主要有快捷键、位图、光标、对话框、图标、菜单及工具栏按钮等。在特殊情况下，还可以添加自定义类型的资源。
- 项目属性：这个菜单项用于设置项目的各种属性，这个功能在以后的开发实践中会经常用到。单击这个菜单项后，会出现如图 2-16 所示的项目属性设置对话框。



图 2-16 设置项目属性对话框

从图 2-16 中可以看到，整个项目属性的设置非常复杂，幸运的是在大多数情况下，只需要使用默认参数就可以了，只有少数几个参数需要进行配置，下面就这些常用的配置做简单介绍。

- (1) 选择属性对应的版本。在设置对话框界面的上方，我们可以通过下拉按钮来选择这个属性配置影响的版本，还可以选择代码运行的平台。通常，我们选择当前活动的是 Debug 版本或 Release 版本，也可以使用配置管理器创建自定义的配置方案，进行更加详细、更加个性化的设置。

知道更多：Debug 和 Release

软件通常分为 Debug 版本和 Release 版本两种。Debug 版本通常又称为调试版本，包含调试信息，未作任何优化，便于程序员在开发过程中调试程序。等到开发完成后，就可以编译 Release 版本发布给用户，所以 Release 版本又称为发布版本，它往往经过了各种优化，以使用户很好地使用。

Debug 版本的软件在单独运行时，往往需要编译器提供一些库文件，Release 版本则不需要。换句话说就是，Release 版本的软件可以在没有安装 Visual C++ 的计算机上正常运行，Debug 版本则不行。

Debug 版本和 Release 版本之间并没有本质的区别，它们只是一组编译选项的集合。可以修改这些选项，得到优化过的调试版本或带跟踪语句的发布版本。

(2) 常规设置。展开左侧树形节点中的“配置属性→常规”，在这个属性配置页面中，可以对项目的一些常规属性进行配置，包括项目的目录、是否需要 MFC 支持、是否需要公共语言运行时支持等。最常用的是设置“MFC 的使用”这一项。

- 1) 在共享 DLL 中使用 MFC：表示把程序中用到的 MFC 类库作为动态链接库，这样编译生成的程序比较小，但是在运行的时候，需要操作系统提供额外的动态库支持。
- 2) 在静态库中使用 MFC：表示把用到的 MFC 类的内容作为静态库加到程序中，这样编译产生的程序可以在任何 Windows 环境下运行，但是程序的体积比较大。
- 3) 使用标准 Windows 库：表示不使用 MFC 类库，仅使用标准的 Windows 类库。

(3) C/C++ 设置。在这个子项中，可以设置很多跟编译相关的参数、自定义编译器的行为。其中，最常用的设置包括以下几点。

- 1) 附加包含目录：在这里，可以设置其他头文件的搜索目录，当在项目中用到其他程序库时，常常用这个配置项目将其他程序库的头文件目录作为附加包含目录，这样编译器就可以方便地找到它所使用的其他程序库的头文件。
- 2) 警告等级：设置编译时警告提示的级别，默认为“Level 3”，但是建议大家将警告提示级别调高到“Level 4”，这样可以发现很多隐藏的错误，提高代码的质量。
- 3) 预处理定义：通过该选项，可以预先定义宏，对参与编译的代码进行控制。例如，可以在这里定义 Debug 宏，表示这个版本是 Debug 版本，代码中使用 Debug 宏控制的代码将参与编译，为调试提供辅助信息。

(4) 链接器。在这个子项中，可以定制链接器的行为。其中，常用的配置选项主要包括

以下几点。

- 1) 输出文件：指定链接器输出的位置和文件名。为了便于查看，需要将生成的程序放到特定的目录下，此时，可以使用这个配置选项。
- 2) 附加库目录：当程序使用附加程序库的时候，往往需要用这个配置选项来指定附加库文件（.lib）所在的目录。这样链接器才能找到这些库文件并完成链接。
- 3) 附加依赖项：任何程序都不是单独存在的，往往需要其他程序库为其提供支持，这就是程序的附加依赖项。在这里，设置项目需要引用的其他程序库文件。

4. 生成菜单

生成菜单中的功能跟构建解决方案有关。如果说前面的那些菜单已经让人眼花缭乱，胃口大开，那么现在真正的主菜出来了。下面来看看这个菜单的具体功能。

- 生成解决方案：构建整个解决方案。
- 重新生成解决方案：重新构建整个解决方案。如果解决方案中有多个项目，那么所有解决方案都将重新生成。
- 清理解决方案：删除编译、链接过程中产生的中间文件和最终编译结果。
- 生成...：以上三项都是针对整个解决方案的，接下来的解决方案针对当前项目的构建，通常是构建当前活动的项目，当然，也可以选择构建解决方案中的某个特定项目。
- 批生成：批量编译生成解决方案，链接不同工程或同一工程的不同设置。例如，可以同时生成某个项目的 Debug 版本和 Release 版本。

知道更多：选择合适的构建方式

构建工程的时候，只对工程中修改过的文件进行增量编译，然后连接生成最终结果。重新构建整个工程时，不管文件是否做过修改，都会编译所有的源文件。

如果修改某个文件，构建后发现修改没有起作用，可以重新构建整个工程，使修改生效。

5. 调试菜单

如果构建的解决方案还存在这样或那样的问题，那么就需要调试修正存在的问题。调试菜单如图 2-17 所示，该菜单提供了 Visual C++ 中与调试相关的功能。调试是开发中最常进行的一项工作，如果说构建是一道主菜，那么调试就可以说是主食了。两者往往相伴而行，构建完成后需要调试，调试完成后需要重新构建。下面来看看调试菜单中的具体功能。

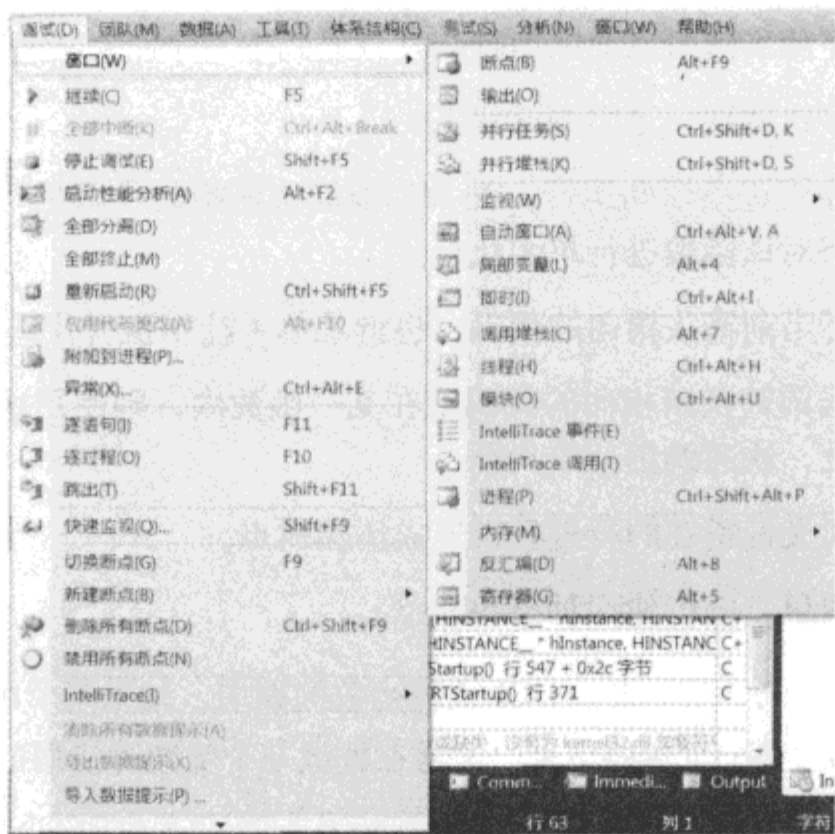


图 2-17 调试菜单

- 窗口：打开跟调试相关的窗口，显示相关的调试信息。
 - 断点：打开断点窗口，显示当前工程中的所有断点。
 - 输出：打开输出窗口，显示构建或者调试过程中的输出信息。
 - 监视：打开监视窗口，可以将调试过程中感兴趣的变量或者表达式添加到监视窗口，监视窗口实时监视这些变量或者表达式的值，反映程序的执行状态和结果。
 - 自动窗口：根据程序当前运行的上下文，自动显示相关变量的值和数据类型。
 - 调用堆栈：显示调用堆栈中的内容，可以从中清楚地查看函数调用关系。自动窗口和调用堆栈窗口如图 2-18 所示。
 - 内存：显示内存窗口。在调试过程中，可以通过内存窗口查看特定内存位置的值。

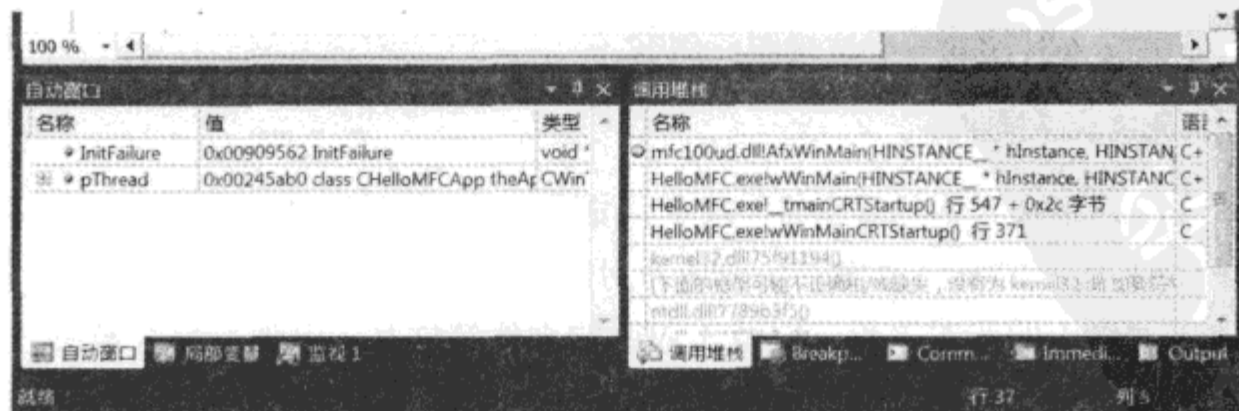


图 2-18 自动窗口和调用堆栈窗口

- 继续：让程序从当前断点继续运行，直到遇到下一个断点或程序结束。
- 停止调试：停止调试，退出调试状态。
- 逐语句：程序将逐条语句运行。如果当前箭头所指的代码是一个函数的调用，使用“逐语句”将对该函数进行单步执行。
- 逐过程：运行当前箭头指向的代码，也就是单步往下执行代码。
- 跳出：如果当前箭头所指向的代码是在某一函数内，则使用这个命令可使程序运行至函数返回处，快速跳出某段代码。
- 切换断点：对光标所在的行进行设置或清除断点。
- 新建断点：这里可以创建两种条件断点。
 - (1) 在函数处中断：这种断点用来监视函数的调用，如果某个函数被调用，则可以在函数内部的某个位置停下来。常常在调试某个函数时使用这种断点。
 - (2) 新建数据断点：这种断点用来监视某个内存位置的值的改变，如果这个内存值发生改变，则停止执行进入调试状态。这种断点在调试比较大的循环的时候非常有用。
- 删除所有断点：删除当前项目中的所有断点。当调试完成后不再需要断点时，可以用这个命令删除项目中的所有断点。
- 禁用所有断点：禁用项目中的所有断点，使断点暂时失效。

2.3.2 Visual C++的常用视图

现在的应用程序已越来越复杂，解决方案中的文件也越来越多，包括各个类的源文件、头文件，还有各式各样的资源文件。如果不对解决方案中的各种文件进行很好的管理，就会陷入文件的汪洋大海中。好在 Visual C++提供了三个管理视图对这些文件进行分类管理，让人有了一览众山小的感觉，需要解决方案中的任何文件，都可以快速找到。这三个管理视图可以帮助我们在复杂的解决方案中快速定位到相关的文件，然后进行查看或编辑，从而提高开发效率。下面就来具体地看看这视图三剑客。

1. 解决方案资源管理器

Visual C++的解决方案往往由多个文件组成，其中包括 C++的源文件、头文件及资源文件等。图 2-19 所示的解决方案资源管理器对这些文件进行了分类。在这个视图中，可以快速浏览解决方案中的所有工程，以及工程中的文件等信息，以便对整个解决方案的文件组织结构有一个清晰的认识。双击某个文件，就可以在编辑窗口中打开这个文件进行编辑，方便快捷。

2. 类视图

对于 C++ 项目而言，最常用的信息恐怕就是项目中各个类的信息了。往往需要查看各个类的成员函数，才能了解这些类的使用方法。在图 2-20 所示的类视图中，可以快速查看工程中所有类的相关信息。单击某个类的名字，会在类视图的下方显示出这个类的所有函数和变量的信息，展开这个类，还可以看到其对应的消息映射表及父类的相关信息等。双击其中的某一项，可以直接定位到相应的代码位置，极大地方便了查找和修改类的代码。在开发实践中，经常会用到这个视图。



图 2-19 解决方案资源管理器

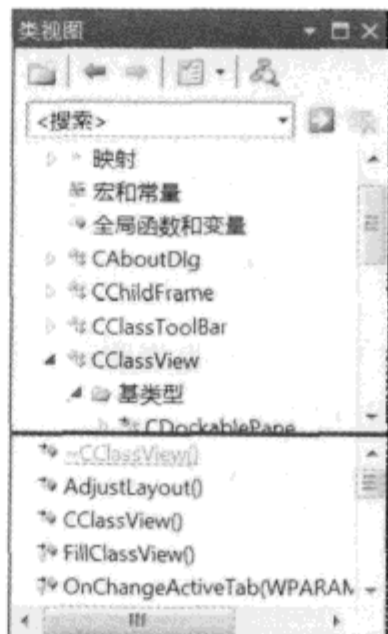


图 2-20 类视图

如果项目中的类太多，则可以在类视图中新建文件夹，把相关的类组织在一起，便于查找。

3. 资源视图

在 C++ 的解决方案中，除了 C++ 的源文件和头文件之外，剩下的就是应用程序会用到的各种资源文件，其中包括位图文件、光标文件、对话框窗体、菜单资源等。虽然资源文件对于普通的 C++ 控制台应用程序而言并不是必须的，但是，对于那些比较复杂的 Windows 程序而言，资源文件却是必不可少的。在图 2-21 所示的资源视图中，可以查看工程中的所有资源。这些资源都按照资源的类型被分类放在不同的文件夹中，可以通过双击其中的某项来打开资源进行编辑。还可以通过鼠标右键菜单来方便地给工程添加新资源。

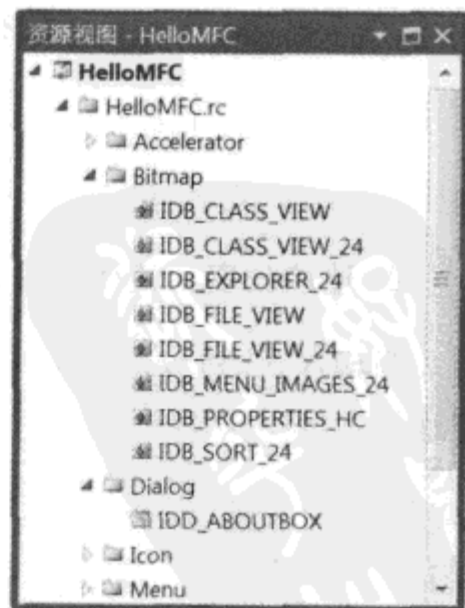


图 2-21 资源视图

2.4 C++世界旅行必备的物品

每个旅行者的背包中都少不了一把瑞士军刀、一瓶云南白药。同样，每个 C++ 世界的旅行者背包中也少不了几款好用的软件工具。

2.4.1 编程助手 Visual Assist

Visual Assist 是一款 Visual Studio 插件，通常也称为 VC 助手，可以用于完成烦琐的工作，提供有用的信息，提高编程的效率。它会让人们感觉到编码不再是一项枯燥的工作，而是一种享受。

Visual Assist 的主要功能包括以下几点。

1. 语法加亮：增加代码的可读性

Visual Assist 可以快速地改变类、函数、变量、预处理宏等元素的颜色，同时提供自定义的功能，用户可以根据自己的喜好为特定的语法元素制定颜色、字体等。语法加亮可以提高程序的可读性。请看图 2-22 和图 2-23 所示的这两段代码的对比。

```
POSITION pos = pDoc->GetFirstViewPosition();  
ASSERT(pos != NULL);  
int nWorkDone = DoSomeWork(pos);  
WaitForSingleObject(m_hEventRecalc, INFINITE);
```

图 2-22 未经过 Visual Assist X 语法加亮的代码段

```
POSITION pos = pDoc->GetFirstViewPosition();  
ASSERT(pos != NULL);  
int nWorkDone = DoSomeWork(pos);  
WaitForSingleObject(m_hEventRecalc, INFINITE);
```

图 2-23 经过 Visual Assist X 加亮后的代码段

很显然，经过语法加亮后，只要看到字符的颜色就知道了这个字符串所表达的语法含义，这样使得代码更容易阅读。

2. 建议列表，使代码输入更迅速

当输入代码时，Visual Assist 会根据用户已经输入的代码进行提示，给出适当的建议供人们选择，如图 2-24 所示。

输入 dispatch 这段代码，Visual Assist 可根据用户的输入给出所有跟 dispatch 有关的函数或者变量，只要单击需要的函数或者变量，就可以快速完成输入。同时，如果用户只记得函数的部分名称，也可以用这个办法找到完整的函数名，而无须查 MSDN。

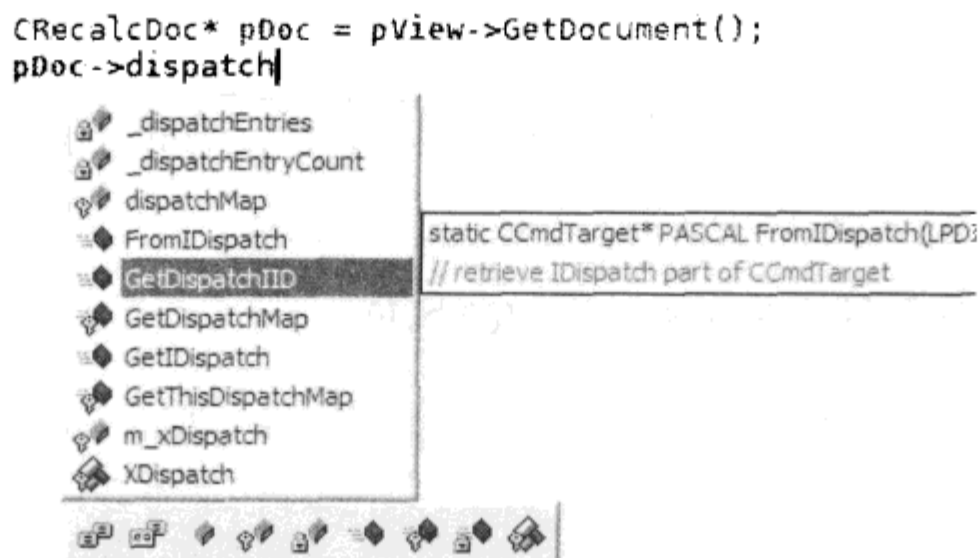


图 2-24 跟上下文相关的建议列表

为了提高编程效率，Visual Assist 还提供了 Autotext 和代码模板的功能，通过快捷键来帮助用户输入常见的代码片段，例如，代码文件的版权声明及固定格式的注释等。

3. 查找和浏览：使得代码之间的跳转更加轻松

编写代码的时候，往往需要在多个代码段之间跳转。Visual Assist 含有最近行为列表，可以在代码的活动部分之间相互转换，就像浏览网页一样可以前进和后退。另外，常常要查看函数是如何实现的，为此 Visual Assist 提供了快速跳转到函数的功能，同时也可以头文件和源文件之间跳转，使得代码之间的导航更加轻松，如图 2-25 所示。

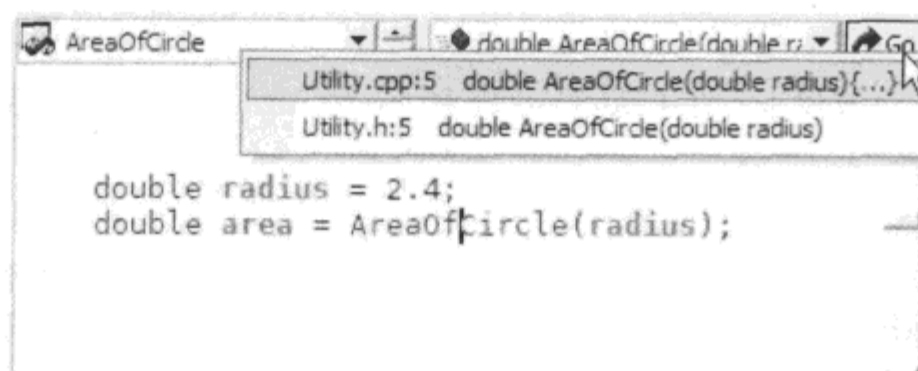


图 2-25 轻松跳转到函数的实现

4. 拼写检查：快速发现错误

Visual Assist 可以在输入代码的同时进行拼写检查，显示跟 Microsoft Word 相似的红色下划线，提示拼写错误，同时还给出修改意见，这下英文不太好的朋友，再也不用担心单词拼写错误而被人笑话了。另外，Spell check code 功能可以检查输入错误的符号，提示这些函数或者变量没有定义，如图 2-26 所示。

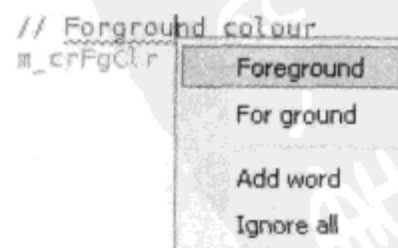


图 2-26 提示拼写错误并给出建议

以上介绍的功能只是 Visual Assist 常用的功能，它还有很多非常有用的功能，比如，代码智能感知、重构、搜索等。它就像一把瑞士军刀，如果仅用它来削个苹果、拧个螺丝，那就太大材小用了。Visual Assist 还有很多功能等着我们去发现、去利用，帮助我们提高编程效率，成为真正的高手。

2.4.2 代码配置管理工具 Visual Source Safe

在实际的开发过程中，常常会遇到这样的情况：试验某个想法、修改文件后，发现想法行不通，这时想让这个文件回到修改之前的样子，但是代码已经被改乱了。这时如果有个时间机器，能够回到过去，取回原来的代码该有多好啊。幸运的是，微软为我们准备了这样的工具——Visual SourceSafe（简称 VSS），它就是我们想要的时间机器。

本质上讲，VSS 就是源代码管理工具，通过它，可以对现有代码进行安全的改动。每次修改代码之前，需要将代码从服务器签出到本地；修改完成之后，再将代码从本地签入到服务器。每次签入、签出都会服务器上留下相应的时间戳，如果发现某次修改引入了新的错误，那么可以将代码回滚到修改之前，就像拥有了一部时间机器，可以让代码回到未被修改之前的状态。同时，它还可以跟踪代码随着用户、项目和时间的变化而经历的更改，方便更新维护代码。

2.4.3 CodeProject 和 CodeGuru

在第 1 章曾经介绍过，学好 C++ 的不二法门就是“多读多写”。多读，就是多阅读别人的代码，他山之石，可以攻玉。在 CodeProject 和 CodeGuru 上有很多非常优秀的代码可以供我们学习和借鉴。

1. CodeProject

CodeProject (<http://www.codeproject.com>) 是一个免费的公开源代码的 Windows 程序设计网站，主要的使用者是 Windows 平台上的程序员。该网站的文章不仅介绍了具体的开发技术，同时附有源代码和例子下载。

2. CodeGuru

CodeGuru (<http://www.codeguro.com>) 也是类似于 CodeProject 的 Windows 程序设计网站，其中集合了大量的技术文章，更有论坛可以供我们提出问题并获得解答。

2.4.4 C++ 百科全书 MSDN

学习一门语言，少不了一本词典。同样，学习 C++ 语言，也少不了一本百科全书。

MSDN 的全称是 Microsoft Developer Network。这是微软公司面向软件开发者的一种信息服务。MSDN 实际上是一个以 Visual Studio 和 Windows 平台为核心的开发虚拟社区，包括技术文档、在线电子教程、网络虚拟实验室、微软产品下载（几乎全部的操作系统、服务器程序、应用程序和开发程序的正式版和测试版，还包括各种驱动程序开发包和软件开发包）、blog、BBS、MSDN WebCast、MSDN 杂志等一系列服务。

作为程序员，接触的更多关于 MSDN 的信息应该是来自于 MSDN Library。它涵盖了微软全套可开发产品线的技术开发文档和科技文献（包括部分源代码），也包括过刊的 MSDN 杂志节选和部分经典书籍的节选章节。在安装 Visual Studio 的时候，可以选择安装脱机的 MSDN Library，但更多时候大家会选择浏览网站上的 MSDN Library，获得最新、最权威的技术资料。在线浏览的地址为 <http://msdn.microsoft.com/zh-cn/library>。



第2篇

欢迎来到〇++世界



在听过了 HelloWorld.exe 的自我介绍，完成了与 C++ 世界的第一次亲密接触后，大家是不是都急不可待地想要一试身手，开始编写 C++ 程序了呢？

程序是使用数据来描述现实世界的。当尝试使用数据来描述现实世界时，马上就遇到了一个问题：C++ 世界中的数据这么多，我们一个都不认识，该从哪里开始啊？别着急，现在就来介绍 C++ 世界的芸芸众生：基本数据类型。

3.1 C++ 中的数据类型

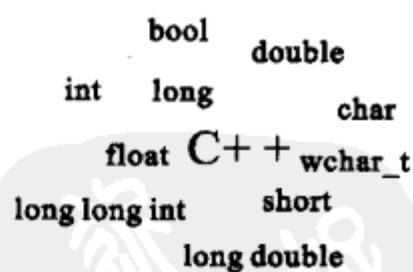
编程是使用程序设计语言来描述和表达现实世界的。现实世界中有很多客观存在的事物，例如，数字、人、车辆等。很多数据是同一类的，比如人的名字都是文字；人的身高都是数字。在程序设计语言中，将这些相同类型的数据抽象成数据结构。数据结构是对现实世界中的某一类数据的描述，例如，可以用 int 来描述某个范围内的整数，可以用 char 来描述所有的字符。更多时候，数据结构表现为某一类数据的类型，我们也把它称为数据类型。在 C++ 世界中，同样数据类型的数据是同一个类别的，也有着相同的一些特征。为了描述现实世界中丰富多样的事物，通常将这些事物定义成具体的数据，而数据类型则是这些事物的种类。数据类型就像 C++ 世界的百家姓，一个数据的数据类型，决定了这个数据是哪一家的人，如图 3-1 所示。

在 C++ 语言中，数据类型可分为基本数据类型和构造数据类型。

1. 基本数据类型

基本数据类型是 C++ 语言中最基础的数据类型，是 C++ 世界最底层的民众，都具有自我说明的特点，不再具有可分性，比如一个整数、一个字符等。

C++ 作为一种高级程序开发语言，它提供了很多常见的数据类型。用这些数据类型，可以声明不同的数据，表示现实世界中的各种事物。例如：



```

      bool      double
    int  long      char
      float C++ wchar_t
long long int  short
              long double
  
```

图 3-1 C++ 的和谐大家庭

```
string strName;    // 姓名
int nHeight;      // 身高
```

2. 构造数据类型

现实世界是复杂的，只使用 C++ 所提供的基本数据类型还不能够完全描述复杂的现实世界。但是我们发现，复杂事物都是由简单事物组成的，一个复杂的事物可以分解成多个简单的事物。在 C++ 中也提供了这样一种机制，可以将多个基本数据类型组合起来，构成一个比较复杂的构造数据类型，描述更加复杂的事物。例如，可以将多个字符组合成一个字符串数组，用来描述复杂的字符串；还可以将两个数值数据组合起来形成一个新的数据类型，用来描述长方形的长和宽。例如，可以这样来创建一个新的构造数据类型：

```
// 长方形数据结构
struct Rect
{
    int m_nLength;    // 长
    int m_nWidth;     // 宽
};
```

一个构造数据类型可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或另一个构造数据类型。在 C++ 中，构造数据类型有 4 种：① 数组类型；② 结构类型；③ 联合类型；④ 枚举类型。

3.2 变量和常量

C++ 世界住满了各种数据量。这些数据量都有自己的数据类型，就像每个人都有自己的姓氏一样。基本数据类型的数据量，按其取值是否可改变分为常量和变量两种。在程序执行过程中，如果数据量的值不改变，程序只能读取而不能修改它的值，那么这样的数据量称为常量；如果数据量的值可以改变，程序既可以读取也可以修改，那么这样的数据量称为变量。例如：

```
// 变量和常量
int nNum = 1;
```

在代码中，有两个数据量：nNum 和 1。其中，nNum 的值可以修改，所以它是一个变量；而 1 不能修改，所以是一个常量。

常量和变量可以与数据类型结合起来，对 C++ 世界的所有“居民”进行分类。例如，可将 C++ 数据分为整型常量、整型变量、浮点常量、浮点变量，等等。在程序中，常量可以直接使用，而变量则必须先声明后使用。

使用数据类型可以描述数据。在程序中，数据类型具体化后就成为程序当中的变量或者常量，用来表示现实世界中的各种事物，成为算法处理的对象。

3.2.1 声明变量

C++世界的居民都有自己的数据类型，数据类型决定了居民的各种特性。可以使用数据类型来定义一个变量。现实世界中的各种数据，可以按照如下的语法格式定义变量：

```
数据类型 变量名;  
数据类型 变量名1, 变量名2, 变量名n;    // 不推荐的形式
```

变量声明由数据类型和变量名两部分构成。数据类型是对变量类型的描述，用于指定数据是整型、浮点型还是自定义数据类型等。变量名是变量的名字，用来标记变量的符号。变量名本身是一种标识符，在给变量命名时需要遵循标识符规则。下面是一些变量声明的示例：

```
int nStudentNum;    // 整型变量，用于表示学生数  
// 定义多个整型变量，分别表示学生的 ID、所在的班级和年龄  
int nStudentID, nClass, nAge;  
float fScore;    // 浮点型变量，用于表示学生的分数  
// 非法的变量声明  
int case;    // 不能使用关键字作为变量名  
int lmember;    // 变量名不能以数字开始  
bool do you love me;    // 变量名中不能有空格
```

以上示例第一句声明了一个整型的变量，变量名是 nStudentNum；第二句同时声明了三个整型变量；第三句声明的是一个浮点型变量。通过上面这几条声明语句，又为 C++ 世界增添了好几位“居民”。

声明变量时，应注意以下几点。

- 允许在一个数据类型说明符后同时声明多个相同类型的变量。各变量名之间用逗号间隔。数据类型说明符与变量名之间至少有一个空格间隔。
- 最后一个变量名之后必须以“;”结尾，表示声明语句的结束。
- 变量声明必须放在变量使用之前，一般放在函数体的开始部分。

3.2.2 给变量取个好名字

一个人如果有个好名字，就很容易给人留下良好而深刻的印象。变量的名字也一样，合适的变量名包含跟变量相关的信息，可以自我解释，让人更容易理解，从而提高代码的可读性。那么如何给变量取一个合适的名字呢？

比较下面三个变量名：

```
int nStudentNum;  
int N;  
int theNumberOfStudent;
```

这三个变量都用来表示学生的数量。如果要问这三个变量名哪个最好，你肯定会说第一个变量最好，因为第一个变量名一看就知道是用来表示学生数量的。

没错！好的变量名可以解释变量所表示的意义，无须过多的注释和文档，整个代码就清晰可读。要为变量取一个好名字，通常要使用某种命名规则。目前业界比较流行的命名规则当属微软公司提倡的“匈牙利命名法”。它的基本原则是：

变量名 = 属性 + 类型 + 对象描述

使用匈牙利命名法可以提高变量的可读性，但其最大的缺点就是烦琐：有些常用的前缀太长，为变量增加意义不大的前缀，增加了额外的负担。

世界上并无所谓最好的命名规则。在实践中，可以根据业界现有的一些共性规则，再结合项目的实际情况来制定一种令大多数项目成员都满意的命名规则，并在项目中贯彻实施，这就是“最好”的变量命名规则了。业界共性的命名规则主要有以下几点。

1. 简单易懂

变量名应当直观，方便拼读，可望文生义。变量名最好采用英文单词或组合，便于记忆和阅读，切忌使用汉语拼音来命名。程序中的英文单词一般不复杂，用词应当尽量做到地道、准确，例如，表示当前数值不要把“fCurrentValue”写成“fNowValue”。

2. 最短长度，最大信息量

通常，编译器对变量名的长度没有限制。一般来说，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符不足为怪。变量的名字是不是越长越好呢？不见得。例如，同样是表示最大值，变量名“maxval”就比“maxValueUntilOverflow”好用。另外，单字符的变量名也是有用的，常见的如i、j、k等，它们通常可用做函数内的局部变量。

3. 变量名由名词构成

变量表示的是现实世界中的一个事物，其本质是一个实体，所以变量名的核心应该是一个名词。变量的名字应当使用“名词”或者“形容词+名词”的形式。例如：

```
float fValue;           // 名词  
float fOldValue;        // 形容词 + 名词  
float fNewValue;        // 形容词 + 名词
```

4. 不要使用数字编号

尽量避免名字中出现数字编号，如“Value1”、“Value2”等，除非逻辑上的确需要编号。

5. 常量大写

常量全用大写字母表示，并且用下划线分割单词。例如：

```
const int MAX = 100;
const int MAX_LENGTH = 100;
```

6. 使用约定俗成的前缀

前缀可以很好地解释变量的属性，让变量的含义一目了然。例如：变量加前缀 s_，表示静态（static）变量；变量加前缀 g_，表示全局（global）变量；类的数据成员加前缀 m_，表示成员（member）变量。

3.2.3 变量初始化

变量声明后，系统就会给每个变量分配内存空间，用于存放对应类型的数据。在使用变量之前，最好能够对其进行初始化。可以通过传递初始值给数据类型的构造函数来完成变量的初始化，或者通过在声明变量的同时赋给初始值对变量进行初始化。例如：

```
string strName("ChenLiangqiao"); // 姓名，使用构造函数初始化，初始值是
    "ChenLiangqiao"
int nHeight = 173;                // 身高，使用赋值进行初始化，初始值是 173
```

另外，我们也可以在变量声明后，通过赋值操作符给变量赋初始值来完成初始化。例如：

```
int nStudentNum;                  // 声明变量
nStudentNum = 1000;               // 赋初始值
```

值得注意的是，虽然变量的初始化工作不是必须的，但是建议对程序中的每个变量进行初始化。这是因为变量声明后，如果不进行初始化，则系统会给定一个随机值，而使用这个随机值进行操作，可能导致程序运行结果出错，甚至程序崩溃。

3.2.4 常量

所谓常量，就是在整个程序运行过程中始终不变的量，简单来讲，就是程序中直接使用的数值、字符、字符串等。常量没有名字，就像 C++ 世界的无名人士，可以无须声明而直接使用。因为其数据只能读取，不能修改，所以通常用来给一个变量赋值或者直接使用。例如：

```
// 用常量对变量赋值
strName = "ChenLiangqiao";
// 直接使用常量进行计算
fArea = fR * fR * 3.1415926;
```

这里的“ChenLiangqiao”和“3.1415926”就是两个常量，分别用来对变量 strName 进行赋值和参与乘法运算，当完成赋值操作和乘法运算后，这两个常量也就不再有意义了。

C++中的常量主要包括整型常数、浮点型常数、字符常量、字符串常量。

1. 整型常数

整型常数就是以文字形式出现的整数。整型常数的表示形式最常见的是十进制，也可以根据需要采用八进制和十六进制。在程序中，可以根据前缀来区分各种进制数。因此在书写常数时不要把前缀弄错而造成结果不正确。

- 十进制整数：例如 0、123、-1 等。
- 八进制数：以 0 开始的整数就是八进制数，例如 0123、-022 等。
- 十六进制数：以 0x 或 0X 开始的数就是十六进制数，例如 0x123、-0X2 等。

示例代码如下：

```
nHeight = 173;           // 十进制常数
nHeight = 0255;          // 八进制常数
nHeight = 0xAD;          // 十六进制常数
```

上面的代码分别采用不同进制形式的数对一个变量赋值。虽然这些常数的表现形式不同，但是它们所代表的数值都是一样的。

2. 浮点型常数

浮点型常数就是以文字形式出现的浮点数，有两种表示形式：小数形式和指数形式。小数形式由数字和小数点构成，如 1.0、0.1、.123、0.0 等。指数形式中 1.1e5 表示 1.1×10^5 ，0.123E-4 表示 0.123×10^{-4} 等。

3. 字符常量

字符常量就是程序中使用的单个字符，如“a”、“A”、“!”等。在 C++世界中，我们使用单引号（'）来表示一个字符常量。例如：

```
// 用一个字符常量对变量赋值
char aMark = 'A' ;
// 输出一个字符常量 '!'，显示到屏幕
cout<<'!'<<endl;
```

除了上述常见的可显示的字符形式外，C++还允许使用一些特殊的字符常量，这些字符是不可显示字符，同时也无法使用键盘输入。这些字符虽然不可显示，但是可以用来表示计算机响铃、换行、回车等一些特殊的意义。这些字符都以“\”开始，表示将“\”后的字符转换成其他的含义，所以这些字符称为转义字符。表 3-1 列出了 C++的常用转义字符。

表 3-1 C++中的常用转义字符

转义字符	意 义
\a	响铃，输出该字符时，屏幕无显示，但计算机喇叭发声，常用来提示用户程序完成某项操作
\n	换行 (n: line)，如果在一个字符串中有这个字符，转义字符后的字符串将换行输出
\t	制表符，输出位置将横向跳格
\r	回车 (return)
\\	输出转义字符 “\” 本身
\"	输出双引号
\'	输出单引号

转义字符的使用跟可显示字符的使用相似，可以把转义字符放到一个字符串中，让它完成相应的功能，也可以直接输出相应的转义字符，例如：

```
// 输出一个换行字符串，这个字符串将被输出为两行
// 恭喜！
// 任务完成！
cout<<"恭喜！ \n 任务完成！ "<<endl;
// 输出一个计算机响铃，提示用户任务完成
cout<<'\a'<<endl;
```

4. 字符串常量

字符串常量就是由一对双引号 (") 括起来的字符序列，如"Hello World!"。注意，因为双引号是字符串的界限符，所以想在字符串中使用双引号，就要使用转义字符来表示。例如：

```
// 使用字符串常量对变量赋值
strName = "ChenLiangqiao";
// 输出字符串常量
// 这里使用了转义字符 “\” 来输出字符串中的双引号，最终输出结果如下
// 你的名字是：“ChenLiangqiao”
cout<<"你的名字是：\"ChenLiangqiao\" "<<endl;
```

3.2.5 用宏与 const 关键字定义常量

常量就像 C++世界的“雷锋”，只做好事而不留名字。但是，我们不能太亏待“雷锋”，也要给常量一个名分。更重要的是，有时候必须要给常量一个名分是必要的。例如，要编写一个有关圆的计算程序，无疑会多次用到 π 常量。那么，是不是一定要在程序中反复多次地写 3.14159 呢？不是，给常量 π 一个名分，就可以解决这个问题。

如果常量也有一个名分，就可以在程序中多次使用常量的名字来代表常量，也就达到了在程序中多次使用同一个常量的目的。给常量一个名分，通常有两种方法：使用宏或者 const

关键字。

在 C++ 中，可以使用 `#define` 预编译指令来定义一个宏：

```
#define 宏名称 宏值
```

其中，“宏名称”就是要定义的宏，通常用一个大写的有意义的名称来表示。“宏值”就是这个宏所代表的常量，它可以是一个常数、一个字符串，甚至是一个更加复杂的语句。比如，可以将 π 值定义为一个宏 `PI`：

```
#define PI 3.14159
```

有了常量所对应的宏，就可以在代码中使用宏来代替常量进行相应的计算。例如：

```
// 计算圆的周长
double fCircle = 2 * PI * fR;
// 计算圆的面积
double fArea = PI * fR * fR;
```

这里，使用了 `PI` 宏代替了原本应该使用的常量 `3.14159`。宏的本质是一种替代。当预编译程序在处理源代码时，如果发现代码中使用了宏，就会用宏的值来代替宏。例如，上面的代码会被分别修改为

```
// 宏替换后的代码
double fCircle = 2 * 3.14159 * fR;
double fArea = 3.14159 * fR * fR;
```

归根结底，还是在代码中多次使用了常量，只是预编译程序帮我们完成了部分烦琐的工作而已。

使用宏给常量一个名分，除了可以避免多次输入重复使用的常量之外，宏还进行超值大赠送，带来了一些额外的好处。

1. 宏让代码更简洁明了

一个宏名称往往比一个简单无意义的常量数字或者常量字符串包含了更加丰富的信息，可以增加代码的可读性；同时，宏比常量更简单，可以使代码更简洁。对比下面两段代码：

```
// 不使用宏的代码
for( int i = 0; i < 100; ++i )
{
    // ...
}
// 使用宏的代码
#define MIN 0
#define MAX 100
for( int i = MIN; i < MAX; ++i )
{
    // ...
}
```

通过对比发现，虽然两段代码实现的功能是一样的，但是给代码读者的信息不太相同。第一段代码只是表示这个循环是从 0 到 100 之间，至于为什么是从 0 到 100，只有自己去猜测了。第二段代码通过宏的使用，明确地告诉了我们这个循环是在一个最小值和最大值之间进行的，这样就可以从代码本身获得更加丰富的信息。

2. 宏让代码更加易于维护

如果需要修改某个多次使用的常量，只需要修改宏的定义就可以了，而不用修改代码中所有的宏。例如，如果想提高圆周率的精度，只需要将宏定义修改为

```
#define PI 3.1415926
```

然后所有使用 PI 宏的地方都会使用这个更高精度的 π 值进行计算。

虽然宏的使用可以带来便利，但是因为它是在预处理的时候进行无条件的替换，并没有明确指定这个常量的数据类型，所以带来便利的同时也容易带来问题。C++使用另一种更加稳妥的方法来代替宏的这一功能，这就是 const 关键字。

变量的值是可以修改的，可以在定义变量的时候加上 const 关键字，让变量的值不可修改，从而成为常量。const 关键字的使用格式如下：

```
const 数据类型 常量名 = 常量值;
```

相比于变量的声明方式，我们只是在数据类型前加了 const 关键字，该关键字告诉编译器这条语句所定义的变量的值是不可修改的，也就是说，该变量可以当成常量来使用。需要注意的是，因为常量的值不可以修改，所以必须在定义常量的同时完成它的赋值。例如：

```
// 定义常量 PI
const double PI = 3.14159;
```

使用 const 的常量一经定义后，就不能再进行修改，否则会产生编译错误。例如，如果在程序中降低 PI 的精度，偷工减料是不行的，例如：

```
// 不能修改 const 常量的值
PI = 3.141;
```

这样给一个 const 修饰的变量赋值是不行的。正是编译器检查变量赋值，使得我们无法对 const 修饰的变量进行赋值，从而也就无法改变这个变量的值。既然变量的值无法改变，那么这个变量也就成了常量。

既然宏和 const 关键字都可以用来给常量一个名分，那么该如何选择呢？到底是用宏还是用 const 关键字？应该更多地使用 const 关键字。比如，要定义 PI 这个常量，可以采用以下两种方式：

```
#define PI 3.14159
const double PI = 3.14159
```


这两种方式在语法上都是合法的，但是第二种方式要比第一种方式好，因为如果使用 `#define` 定义宏，PI 会在代码的预编译阶段被预编译处理器替换成 3.14159，宏的名称不会出现在符号表中，这样会给代码后期的调试带来麻烦，可能会遇到一个数字，却不知道它从何而来，这就是我们常说的 Magic Number。使用 `const` 定义，既可以保证 PI 值的唯一性，又便于调试，同时还可以对数据类型进行检查，借助编译器来减少错误的发生。所以，我们总是优先使用 `const` 关键字来定义常量。

3.3 数值类型

每天睁开眼睛我们都能看到各种数字：从表示上班时间的 9 点到表示白菜价格的 3 角 5 分，从房价到空气污染指数等，我们生活在一个数字的世界中。为了描述这个数字世界，C++ 提供了丰富多样的数值类型，从整数到小数、从有符号数到无符号数。有了这些数值类型，C++ 就可以表示现实的数字世界。

3.3.1 整型数值类型

整型数值是 C++ 中最常用的数值数据类型，它用来表示现实世界中的各种范围的整数。按照数据类型所表示的取值范围，整型数值类型又可以分为以下几种。

1. 基本型

其类型说明符为 `int`，在内存中占 2 个字节，其取值为基本整型常数。

2. 短整型

其类型说明符为 `short` 或 `short int`，所占字节和取值范围均与基本型的相同。

3. 长整型

其类型说明符为 `long` 或 `long int`，在内存中占 4 个字节，其取值为长整型常数。

4. 长长整型

为了应对 64 位下的开发，C++0x 引入了可以表示更大范围的长长整型，它的类型说明符是 `long long` 或 `long long int`，在内存中占 8 个字节。

另外，还可以用关键字 `unsigned` 或者 `signed` 对这些整型数据类型进行修饰，构成无符号或者有符号的整型数据类型。各种无符号类型量所占的内存空间字节数与相应的有符号类型量的相同。但由于无符号类型量省去了符号位，故不能表示负数。表 3-2 列出了 C++ 中各种整型数值的数据类型、取值范围、字节数和使用规则。

表 3-2 整型数值类型

数据类型	字节数	取值范围	使用规则
int	16	-32768 ~ 32767	最常用的整型数值类型，用来表示常用的整型数值
signed int	16	-32768 ~ 32767	跟 int 一样，这种类型可以用 int 代替
unsigned int	16	0 ~ 65535	无符号的 int，只能表示正整数。当要表示某些数据只有正数值的时候，可以使用无符号数
short int	16	-32768 ~ 32767	short 类型在使用上可以等同 int 类型
signed short int	16	-32768 ~ 32767	—
unsigned short int	16	0 ~ 65535	—
long int	32	-2 147 483 648 ~ 2 147 483 647	如果要表示更大范围的整数，就用 long
signed long int	32	-2 147 483 648 ~ 2 147 483 647	—
unsigned long int	32	0 ~ 4 294 967 295	—
long long int	64	9223372036854775807 ~ -9223372036854775808	比 long 还要 long 的整数，我们不用去数它到底有多少位，只需要知道，当要表示很大很大的整数时，就可以用它

3.3.2 浮点型数值类型

数字除了整数外，还有小数。在 C++ 中使用浮点型数值类型来表示小数。根据取值范围的不同，C++ 中的浮点型数值类型可以分为单精度型、双精度型和长双精度型。

1. 单精度型

其类型说明符为 float。单精度型浮点型数值占 4 个字节内存空间，其数值范围为 $3.4\text{E}-38 \sim 3.4\text{E}+38$ 。

2. 双精度型

其类型说明符为 double。双精度型浮点型数值占 8 个字节内存空间，其数值范围为 $1.7\text{E}-308 \sim 1.7\text{E}+308$ 。

3. 长双精度型

其类型说明符为 long double。长双精度型浮点型数值占 10 个字节内存空间，其数值范围

为 $3.4\text{E}-4932 \sim 1.1\text{E}+4932$ ，这种类型通常用于科学计算中。

表 3-3 列出了浮点型数值的数据类型、字节数、取值范围和使用规则。

表 3-3 浮点型数值类型

数据类型	字节数	取值范围	使用规则
float	32	$3.4\text{E}-38 \sim 3.4\text{E}+38$	如果要表示的小数数值不是特别大，则使用 float 可以帮助节省空间
double	64	$1.7\text{E}-308 \sim 1.7\text{E}+308$	要表示通常遇到的小数，double 够用
long double	80	$3.4\text{E}-4932 \sim 1.1\text{E}+4932$	可以表示很大很大的数值

3.4 布尔类型

在 C++ 中，布尔类型是一种比较特殊的数据类型，它只可以赋予值 true 或者 false，分别表示逻辑的真与假、是与非。布尔类型的说明符是“bool”。C++ 标准并没有指定布尔类型数据的长度，在 Visual C++ 中布尔类型占 1 个字节。例如：

```
// 布尔类型变量
bool bSuccess = true;
```

布尔类型的对象也可以看成是一种整数类型的对象。当表达式需要一个算术值时，布尔类型对象将被隐式地转换成 int 类型，成为一个整型对象。布尔类型变量转换为整型对象后，如果变量的值是 false，则转换后就是 0；如果变量的值是 true，则转换后就是 1。例如，下面这段程序统计某个范围内的偶数个数。

```
// 判断一个数是不是偶数
// 如果是偶数，则返回 true，否则返回 false
bool isEven( int i )
{
    return i % 2 == 0;
}
// 表示某个数是否是偶数
bool bEven;
// 总的偶数个数
int nEvenCount = 0;
// 统计范围
int nFrom = 0;
int nTo = 100;
// 统计范围内的所有数
for( int i = nFrom; i <= nTo; ++i )
{
    // 调用 isEven() 函数判断当前数值
    // 如果是偶数，则返回 true，否则返回 false
    bEven = isEven( i );
}
```

```

// 将布尔类型变量和整型变量进行运算
// 布尔类型变量将被隐式地转换为整型对象
// 如果 bEven 的值是 true，表示这是一个偶数，相应的统计数也加 1
nEvenCount += bEven;
}
// 输出结果
cout<<nFrom<<"到"<<nTo
    <<"之间的偶数个数为："<<nEvenCount<<endl;

```

在这段代码的循环统计中，首先使用 `isEven()` 函数来判断一个数是否是偶数，如果是偶数，则返回 `true` 值给 `bEven` 赋值。然后将这个布尔类型变量和 `nEvenCount` 整型变量进行累加运算，布尔类型会被隐式地转换成整型。如果 `bEven` 的值是 `true`，转换之后是 1，正好表示为偶数，就给总数加 1；正好表示为奇数，就给总数加 0。通过 `bEven` 和 `nEvenCount` 的累加，就很好地完成了偶数的统计功能。

另外，在使用布尔类型的时候还要注意，用任何非 0 数给布尔类型变量赋值，其值都为 1。0 和 1 构成了布尔类型的取值范围。例如：

```

bool a = 2;           // a = 1
bool b = -4;          // b = 1
bool c = a + b;        // a + b = 2, c = 1

```

3.5 字符串类型

除了基本的数值数据外，C++ 中处理得最多的就是字符串数据了。C++ 有两种跟字符串相关的类型：字符类型和字符串类型。

3.5.1 字符类型

我们在学习英语的时候，都是从 ABC 开始学习的，然后才能说一句完整的话。要想让 C++ 也能说话，也得先从 ABC 开始学起，也就是用字符类型数据来表示单个字符。在 C++ 中，字符类型的类型说明符是“`char`”，它占用 1 个字节，取值范围是 -128~127。例如：

```

// 定义字符类型变量，并用字符常量对其进行赋值
char cA = 'A';
char cB;
cB = 'b';
// 输出字符
cout<<cA<<endl;

```

跟整型数值一样，字符类型也可以受到 `unsigned` 关键字的修饰，形成无符号的字符类型。表 3-4 列出了字符类型的数据类型、字节数、取值范围和使用规则。

表 3-4 字符型数值类型

数据类型	字节数	取值范围	使用规则
char	1	-128 ~ 127	用来表示各种字符
signed char	1	-128 ~ 127	—
unsigned char	1	0 ~ 255	实际上, char 可以当做一种取值范围更小的整型数值类型来使用

char 字符类型取值范围有限, 只能用来表示有限的 ANSI 字符。如果要表示使用广泛的 UNICODE 字符, 例如中文字符, char 就鞭长莫及了。为了表示 UNICODE 字符, C++ 提供了另外一种字符类型 wchar_t, 它占用 2 个字节, 可以表示更大范围的字符。例如, 可以通过下面的方式来输出一个中文字符:

```
// 定义一个 wchar_t 类型字符变量
// 并用一个中文字符对其赋值
wchar_t cChinese = L'中';
// 设置 wcout 输出对象并输出中文字符
wcout.imbue ( locale ( "chs" ) );
wcout<<cChinese<<endl;
```

3.5.2 字符串类型

在学会了 ABC 之后, 就可以开始学习 “Good Morning” 了。这就是一个由多个字符组成的字符串。C++ 并没有一个单独的所谓字符串类型。为了表示一个字符串, 通常用 STL 中定义好的字符串类型 basic_string 来表示字符串。这种字符串类型实质上是对一个字符数组的包装。在字符数组中, 每个位置保存一个字符, 用 “0” 表示字符串的结束。如果包装的是 char 类型的数组, 则字符串类型为 string。同样, 如果包装的是 wchar_t 类型的数组, 则字符串类型就是 wstring。例如:

```
// 定义一个 string 类型变量, 表示英文字符串
string strMorning("Good Morning!");
// 定义一个 wstring 类型变量, 表示中文字符串
wstring strChina(L"中国");
// 输出字符串变量
cout<<strMorning;
wcout.imbue( locale ( "chs" ) );
wcout<<strChina<<endl;
```

从以上代码中可以看到, 因为编码方式的不同, 不同的字符串在输出的时候需要采用不同的方式。对于 wstring 类型的字符串变量, 在输出的时候需要使用 wcout 对象, 并且需要用 imbue() 设定字符的编码方式。

string 类型不仅包装了字符数组, 同时还提供了很多跟字符串相关的操作, 例如, 可以获

得一个字符串的长度，或者在字符串中查找某个字符，等等，极大地方便了对字符串的处理。例如：

```
string strMorning("Good Morning!");  
// 通过 length() 函数得到字符串的长度  
int nLength = strMorning.length();  
// 在字符串中查找字符  
string::size_type nPos = strMorning.find('o');  
// 输出结果  
cout<<strMorning<<"的长度是："<<nLength  
    <<"\n 在位置"<<nPos<<"有一个字符 o。"<<endl;
```

以上代码通过使用字符串类型所提供的 length() 函数得到字符串的长度。通过 find() 函数可以在字符串中查找某个字符或字符串，它得到的是这个字符在整个字符串中的位置。很多时候，编程都涉及对字符串类型的数据进行处理，为了满足这种需要，字符串类型支持很多函数，将在后面的章节中做详细介绍。

3.6 数组

现在已经可以使用基本数据类型来描述这个现实世界了。你看，陈良乔同学刚学完这些知识，就被应聘到一个小公司做程序员了，但当然是个“冒牌”的。上班第一天，老板给他的任务就是写一个工资查询程序。因为公司实在小，只有老板、老板娘和他自己，所以任务比较简单。这位“冒牌”的程序员坐在电脑前，运指如飞，很有“高手”的感觉，一会工夫就完成了任务。

```
int _tmain(int argc, _TCHAR* argv[])  
{  
    int nSalaryCEO = 0;           // 老板的工资  
    int nSalaryCXO = 99999;      // 老板娘的工资  
    int nSalaryEmployee = 2000;  // 陈良乔的工资  
  
    // 处理工资，对工资进行查询 ...  
  
    return 0;  
}
```

完成后，老板试了试，输入员工的序号，果然能够马上查到员工的工资，如果输入错误，还有提示。老板觉得不错，对小陈嘉奖了一番。

一段时间后，公司发展了，员工数增加到了 100 人，老板让小陈把程序改为可以查询 100 个员工的工资。小陈不敢懈怠，一上午不停地执行 Ctrl+C 加 Ctrl+V 操作，为每个员工都定义一个变量保存其工资，忙了一上午才完成程序。又过了几年，公司发展成一家大型跨国公司了，员工数达到了 100 000 个。这时，老板让小陈升级以前的工资查询程序，实现查询

100 000 个员工的工资的功能。想想几年前的经历，想象那长长的代码，这次真是遇到了艰巨的任务。

使用基本数据类型，只能定义单个数据来表示单个的事物，例如，上面例子中的各个员工的工资。但是，现实世界中往往相同类型的数据有很多，例如，一个学校所有学生的成绩，一个公司所有员工的工资，等等。这些数据的数据类型相同，并且形成一个相关的序列。针对这种情况，可以使用数组这种构造型数据类型来表示，将这些相关且相同类型的数据集合起来，让这些数据排排坐，吃果果。

3.6.1 数组的声明与初始化

我们常常遇到这样的数据：数量很大；有相同的数据类型；有相同的处理方式。例如，一个公司所有员工的工资、一个学校所有学生的成绩、一个地区一年的气温，等等。为了描述这种量大且相似的数据，C++提供了数组这种构造型数据类型。

如果把整个内存看成是一座宾馆，那么可以把数组看成是某一层楼上的一个个连续的小房间。这些小房间具有相同的容量，可以存放相同数据类型的数据。当然，房间容量的不同或者连续房间个数的不同，可以在内存中形成不同的数组，如图 3-2 所示。

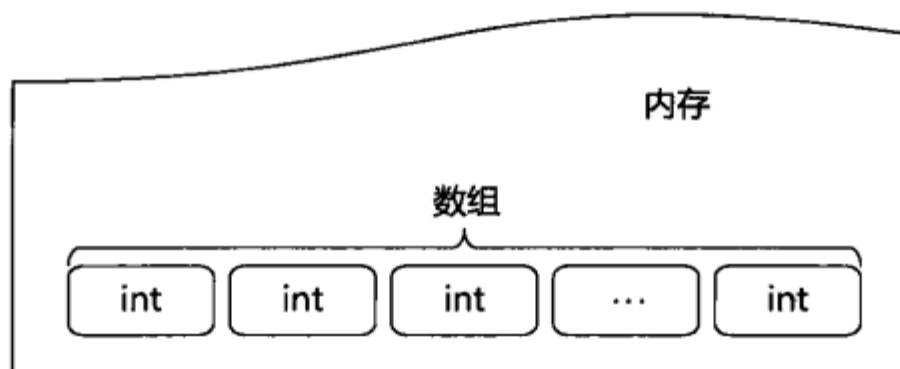


图 3-2 数组就是内存中的多个小房间

在 C++ 中，声明一个数组的语法格式如下：

数据类型 数组名 [个数常量] [个数常量] ...;

在声明数组的语法格式中：数据类型表示这一系列数据的类型，也就是每个小房间的容量，它能够保存什么样的数据；数组名就是给数组的标识符，需要使用数组名来访问数组中的数据，数组名同时也是一个常量，可以表示数组元素在内存中的首地址；个数常量则表示这一系列数据的个数，也就是房间的个数，个数常数必须是 unsigned int 类型的整数，并且必须是常数；“[]”用于确定数组的维数。在数组名后有几个“[]”就表示这是一个几维数组。例如，一个平面的点需要二维数组来表示，空间的点需要三维数组来表示。

下面是一些常见的数组声明形式：

```
int nSalary[100];    // 这是一个一维数组，表示有 100 个整型数可以保存在这个数组中
float fArray[2][3];  // 这是一个二维数组，表示有 2×3 个元素，每个元素的类型都是浮点型
int point[100][100][100]; // 这是一个三维数组，表示有 100×100×100 个元素，每个元素的类型都是整型
```

在定义数组的时候，也可以同时对数组进行初始化。例如：

```
// 定义数组并进行初始化
int nArray[5] = { 1, 2, 3, 4, 5 };
```

这条语句定义了一个长度为 5 的整型数组 nArray，并且分别把 1、2、3、4、5 赋值给数组中的 5 个元素。当然，也可以仅对数组的部分元素赋值，例如：

```
int nBigArray[100] = { 10, 23, 542, 33, 543, 87 };
```

这条语句仅对有 100 个元素的 nBigArray 数组中的前 6 个元素给定了初始值，没有明确给出初始值的元素都是相应数据类型的默认初始值。

对于多维数组，也可以采用同样的方式进行初始化，例如：

```
float fArray[2][3] = { { 2.4, 3.6, 8.3 }, { 2.6, 5.7 } };
```

这条语句中的 fArray 是一个二维浮点数数组，一共有 2 行 3 列。多维数组的初始化以行为单位，每 3 个数构成一行。第 2 行少了最后一个数，是一个不完全的初始化，系统会将这个数初始化为浮点数的默认初始值。

3.6.2 数组的使用

定义好数组后，可以引用数组中的数据来进行运算。若想到某个房间找人，就需要知道他的房间号。同样，要引用数组中的数据，也需要给定数组的下标，也就是数据在数组中的序号。例如：

```
// 定义一个长度为 100 的整型数组
int nSalary[100];
// 通过给定数据在数组中的序号
// 读取 nSalary 数组中的第 25 个元素
int n = nSalary[24];
```

其中，nSalary 是定义的数组名，24 表示想要引用的数据是这个数组中的第 25 个数据。这条语句的含义就是把 nSalary 数组中的第 25 个数据赋值给变量 n。为什么要用下标 24 来引用数组中的第 25 个数据呢？这是因为 C++ 数组的下标总是以 0 开始的，也就是说，nSalary[0] 是这个数组中的第 1 个元素，依此类推，下标 1 表示第 2 个元素，下标 24 就表示第 25 个元素了。另外，需要注意的是，下标必须小于数组定义时的大小。简单来讲，一个长度为 n 的数组，其下标最大值是 n-1，如图 3-3 所示。例如，nSalary[99] 是合法的，表示这个数组的最后

一个元素，而 `nSalary[100]` 就是非法的。当在程序中使用 `nSalary[100]` 访问数组时，会访问到数组以外的内存区域，这会产生非常严重的错误。初学者需要谨记这一点，以免一点小问题造成程序崩溃。

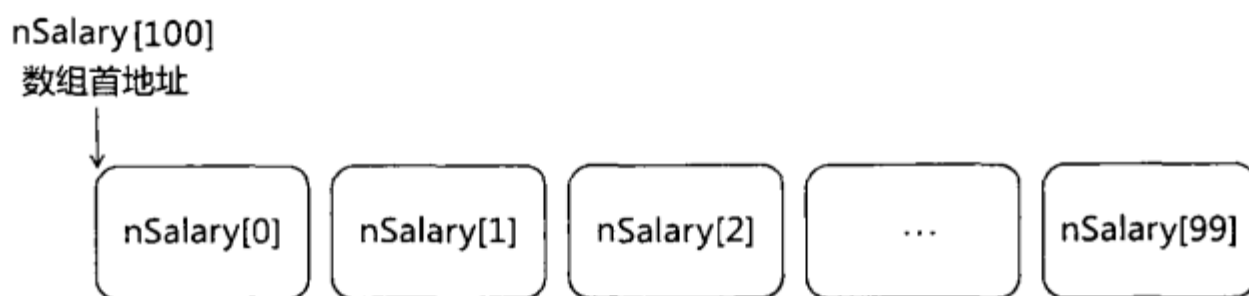


图 3-3 通过下标访问数组

对于二维数组、三维数组等多维数组，同样可以通过给定多个下标来引用数组中的元素。例如：

```
float fA = fArray [1][1];
```

这条语句访问的就是 `fArray` 多维数组中的第 2 行第 2 列的元素。

针对多维数组，C++ 是按照维数从高到低的顺序来排列数据的，较高的维数总是得到优先排列，而在同一维，则按照下标从低到高的顺序进行排列。如图 3-4 所示，C++ 在进行二维数组的内存排列时，先排列第一维的第一层，这一层将包含第二维的所有数据，也就是所有第一维的下标相同的所有数据都会被排布在同一层。例如，`fArray[0][0]`、`fArray[0][1]` 和 `fArray[0][2]` 都会被排布在第一层。在同一层中，所有第一维下标相同的数据再按照第二维的下标从低到高的顺序进行排列，例如，`fArray[0][0]` 将排布在 `fArray[0][1]` 之前。完成这一层的排列后，再进行第一维的第二层排列，依此类推。按照这样的方式来理解数组，通过下标来引用数组中的元素就比较清晰了。

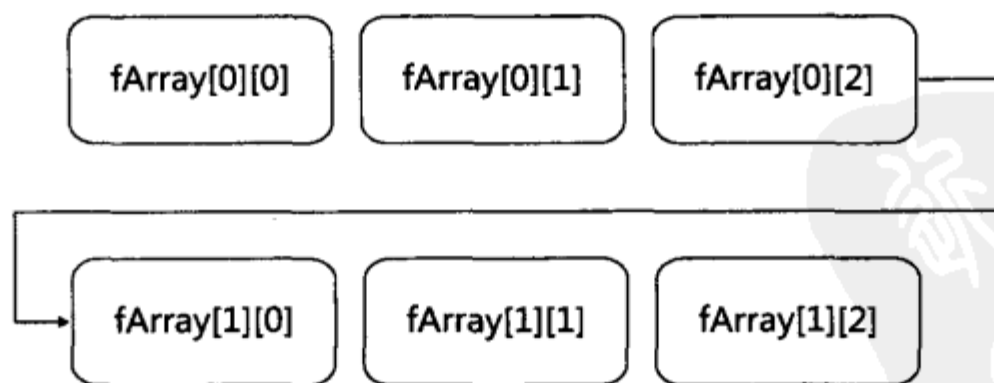


图 3-4 多维数组的存放顺序

通过对数组元素的引用，不仅可以读取它的值，还可以修改它的值。例如：

```
// 对数组中的元素进行赋值  
nSalary[24] = 1200;  
fArray [1][1] = 3.54;
```

通过对数组中元素的引用，可以像使用一个单独的变量一样使用数组中的每个元素。这样在描述大量相似数据的时候，就无须定义多个变量，只需要一个数组就可以了，然后通过不同的下标就可以访问到不同的数据，就像拥有多个变量一样。

陈良乔用一个数组就可完成他的艰巨任务了。

3.7 枚举类型

在现实生活中，常常会遇到这样的数据：一道选择题的答案只有 A、B、C、D 四个选项；一周只有星期一、星期二……星期日七天；一个人的性别只能是男性或者女性。这种数据都只有有限的几种可能值，只能选择其中的某一个。为了定义这种数据，C++ 提供了枚举机制。使用枚举机制定义一种新的数据类型，列举出它的所有可能值。当使用这种新的数据类型定义一个变量时，这个变量的取值也就限定在了枚举类型所列举出的可能取值范围内。声明一个枚举类型的语法格式如下：

```
enum 枚举类型名
{
    // 可能的枚举值列表
};
```

其中，枚举类型名是所要创建的新数据类型的名字。在枚举类型的定义中，可以逐个列出这个枚举类型的可能值。例如，可以将描述一周 7 天的枚举类型声明如下：

```
// 一周枚举类型
enum Weekday
{
    mon,
    tue,
    wed,
    thu,
    fri,
    sat,
    sun
};
```

这样，就可以用 Weekday 这种自己定义的类型来表示一周的 7 天，而它的取值只能是枚举类型定义中列出的某一个。例如：

```
// 定义一个变量表示一周中的某一天，
// 并将其初始值设置为星期天
Weekday nDay = Weekday::sun;
```

枚举类型的这种特殊性，使我们在应用枚举类型时需要注意以下几点。

1. 枚举类型有默认的值

枚举类型实质上是一个整型数值，每个枚举元素都代表一个整数。如果在定义枚举类型时没有给枚举元素赋值，那么它们会有一个默认的整数值，依次是 0、1、2、3……例如，上面例子中的 `Weekday::mon` 的值就是 0，而 `Weekday::tue` 的值就是 1，依此类推。如果认为枚举元素的默认值不合适，也可以在声明的时候另行指定各个元素的数值，例如：

```
// 重新指定枚举元素的数值
enum Weekday
{
    mon = 1,
    tue,
    wed,
    thu,
    fri,
    sat,
    sun = 0
};
```

经过人为地指定枚举元素的数值后，`Weekday::mon` 的值就变为了 1，后面的元素依次加 1，到 `Weekday::sat` 就是 6。因为 `Weekday::sun` 的特殊性，我们又人为地将其指定为 0。

2. 枚举类型变量的赋值

如果变量是枚举类型，就只能使用这种枚举类型的某个可选值对其进行赋值。例如：

```
// 定义一个枚举类型的变量
Weekday nDay;
// 用枚举类型的可选值对枚举类型变量进行复制
nDay = Weekday::sun;
// 尝试用其他数值对枚举类型变量赋值会产生编译错误
nDay = 7;
```

换句话说，如果想把某个变量的取值限定在某几个可选值范围之内，则可以把这个变量声明为枚举类型。

3. 枚举元素的数值是常量，定义后不可改变

在定义枚举类型时，就已经完成了枚举元素数值的定义，数值不是默认数值就是给定的特殊数值。枚举类型的定义完成后，各个枚举元素的数值就成为了常量，要按照常量来处理，不能对其进行赋值。也就是说，不能在完成枚举类型的定义后再改变其中某个元素的值。例如，下面的语句是非法的：

```
Weekday::mon = (Weekday)0; // 尝试改变枚举元素的值，导致编译错误
```

枚举类型实质是一些整型常量。在上面的例子中，完全可以使用 7 个整型常量来表示一周中的 7 天。但是，为什么要选择枚举类型来表示呢？相比整型常量，枚举类型有明显的优势。使用枚举类型可使得代码更易于维护，有助于确保给变量指定合法的期望值，使得代码

更清晰。枚举允许用描述性的名称表示整数值，而不是用含义模糊的整数来表示，所以在可以的情况下，应该尽量使用枚举类型。

3.8 用结构体类型描述复杂的事物

利用 C++ 本身所提供的基本数据类型只能表示一些简单的事物，比如数字、字符串等。现实世界是复杂的，如果仅使用基本数据类型，不足以描述复杂的现实世界。例如，无法使用一个基本数据类型来描述一个长方形，因为它有长和宽两个属性；更无法使用一个基本数据类型来描述一个人，因为他不仅有姓名，还有身高、年龄和性别等属性。虽然可以使用基本数据类型来描述事物的单个属性，但是没法使用一个基本数据类型来描述事物的整体。这时就需要将描述单个属性的基本数据类型组合起来，形成新的构造型数据类型——结构体，用以描述复杂的事物。

3.8.1 结构体的定义

到目前为止，我们所接触过的数据类型都只是单一的，只能描述某个事物的某个方面。比如描述一个人的身高用浮点数类型，描述一个人的名字用字符串类型，描述一个人的性别用枚举类型。在现实世界中，一个事物往往是复杂的，是多个属性的组合。比如，描述一个人的时候，就需要描述其姓名、性别、年龄、身高，等等。当要描述一个比较复杂的事物时，往往需要将多种数据组合起来。为了完成将多个数据打包，C++ 提供了结构体这种机制，即通过把描述事物单个属性的多种数据类型组合在一起来描述更加复杂的事物。在 C++ 中，定义一个结构体的语法格式如下：

```
struct 结构体名
{
    数据类型 1    成员名 1;
    数据类型 2    成员名 2;
    :
    数据类型 n    成员名 n;
};
```

其中，struct 关键字表示要创建一个结构体，结构体名就是要创建的新结构体的名字，通常使用结构体描述的事物作为结构体的名字。在结构体的内部，分别使用多个不同数据类型的变量表示复杂事物的各个属性。因为这些变量共同组成了结构体，所以这些变量称为结构体的成员变量。有了结构体，就可以在结构体中定义多个不同类型的成员变量，通过各个属性来描述一个复杂的事物。例如，可以这样定义描述人这个复杂事物的结构体：

```
// 定义结构体 Human 描述人这个复杂事物
struct Human
```

```

{
    string  m_strName;    // 姓名
    bool    m_bMale;      // 性别
    int     m_nAge;       // 年龄
    int     m_nHeight;    // 身高
    float   m_fWeight;    // 体重
};

```

以前是用各个基本数据类型的变量来分别描述一个复杂事物的各个属性的。这里是将变量集合在一起，打包成一个结构体，如图 3-5 所示。有了结构体，就可以定义一个统一的结构体变量来描述一个具体的复杂事物，代替原来定义多个变量描述同一个事物。例如：

```

// 定义一个 Human 结构体变量描述陈良乔这个人
// 这个结构体包含了他的姓名、性别和年龄等信息
Human chenliangqiao;

```

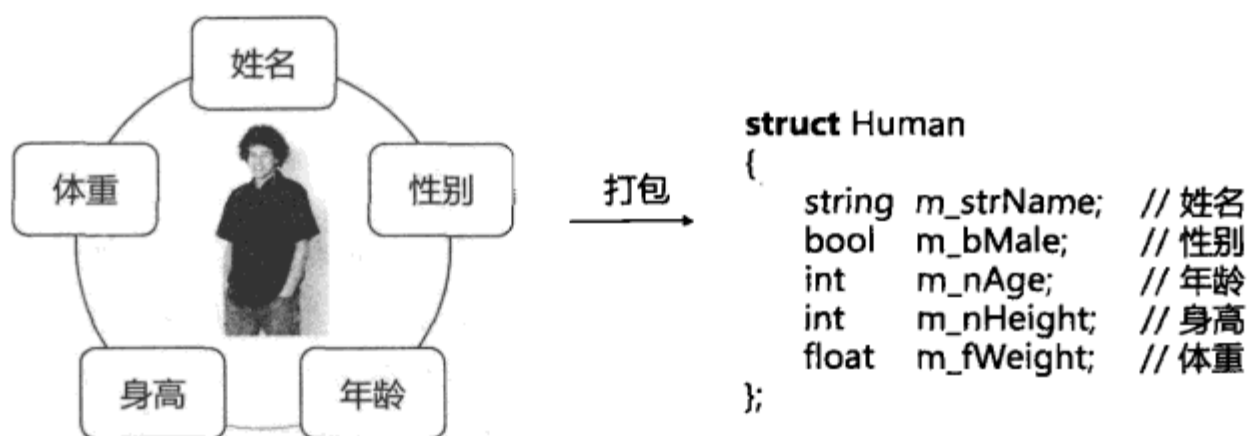


图 3-5 将复杂事物打包成结构体

3.8.2 结构体的使用

结构体变量实际上是结构体中多个成员变量的组合，它包含了结构体中的所有成员变量。有了结构体变量，还需要访问其中的各个成员变量，从而完成对这个事物的具体描述，比如将其表示名字的变量设置为 ChenLiangqiao，将其表示年龄的变量设置为 28，等等。在 C++ 中，使用“.”运算符来引用一个结构体中的各个成员变量，其语法格式如下：

结构体变量.成员变量

其中，结构体变量表示使用结构体定义的变量，而成员变量表示在定义结构体时给这个结构体设定的成员变量。通过对结构体成员变量的引用，可以读取或者设定一个结构体的各个成员变量的数值，从而完成对一个复杂事物的描述。例如：

```

// 对成员变量进行赋值，完成对复杂事物的描述
chenliangqiao.m_strName = "ChenLiangqiao";
chenliangqiao.m_bMale = true;
// 读取成员变量的值
int nAge = chenliangqiao.m_nAge;

```

有了结构体，就可以描述更加复杂的事物，解决更加复杂的问题。以前面的工资管理程序为例，现在老板来了一个新要求——要求不仅能够对员工的工资进行查询，而且还能够对员工的姓名、性别、年龄和工资等人事信息进行查询。虽然新的要求的工资查询系统比简单的工资查询系统复杂了很多，但是我们同样能够轻松应对，因为有了结构体。使用结构体，可以把描述员工的姓名、性别、年龄和工资等多个属性打包成一个结构体类型，从而事情就变得简单多了。

```
// 将员工这个复杂事物的多个属性打包成一个结构体
struct Employee
{
    string    m_strName;        // 姓名
    bool      m_bMale;          // 性别
    int       m_nAge;           // 年龄
    int       m_nSalary;        // 工资
};

// 定义一个常量表示最大员工数
const int MAX_COUNT = 1000;

int _tmain(int argc, _TCHAR* argv[])
{
    // 创建一个结构体数组，管理多个结构体变量
    Employee arrEmployee[MAX_COUNT];

    cout<<"请输入员工信息"<<endl;
    // 当前员工数
    int nCount = 0;
    do
    {
        // 接收用户输入
        // 将用户输入的数据保存到结构体变量的各个成员变量
        cin.clear();
        cin>>arrEmployee[nCount].m_strName
            >>arrEmployee[nCount].m_bMale
            >>arrEmployee[nCount].m_nAge
            >>arrEmployee[nCount].m_nSalary;

        // 检查是否输入完毕
        if ( "end" == arrEmployee[nCount].m_strName )
            break;
        // 开始输入下一个员工的信息
        ++nCount;
    } while( nCount < MAX_COUNT );

    // 显示所有员工的信息
    cout<<"员工信息: \n 姓名\t 性别\t 年龄\t 工资"<<endl;
    for( int i = 0; i < nCount; ++i )
    {
        // 访问数组中结构体变量的成员变量
```

```

        // 获得“员工”这个复杂事物的各种属性
        cout<<arrEmplyee[i].m_strName<<"\t"
             <<arrEmplyee[i].m_bMale<<"\t"
             <<arrEmplyee[i].m_nAge<<"\t"
             <<arrEmplyee[i].m_nSalary<<endl;
    }
    return 0;
}

```

在这段代码中，通过结构体的使用，原来非常复杂的多个变量简化成了一个包含多个成员变量的结构体；原来对多个变量的操作简化为对一个结构体成员变量的操作。这种简化，正体现了结构体在 C++ 中的作用——将复杂的多个事物打包成单个事物，化繁为简。

3.9 指向内存位置的指针

它究竟是神仙的化身？还是地狱的使者？没人知道。但是可以肯定，每个人都给它一个称号——指针。

指针，是 C++ 从 C 语言中继承过来的一种低级数据结构，它提供了一种简便、高效的方式来访问内存。特别是当访问的数据量比较大时，通过指针直接访问数据量所在的内存，可以起到四两拨千斤的效果。正确地使用指针，可以写出更加紧凑、更加高效的程序代码，处理更加复杂的数据结构，例如，链表、二叉树、图等。从这个角度讲，指针成为了 C++ 语言最具吸引力的一个特性，它就是神仙的化身。但是，如果指针使用不当，就很容易产生严重的错误，并且这些错误还具有一定的隐藏性、不易被发现，成为千千万万程序员痛苦的源泉。从这个角度讲，它就是来自地狱的使者。学好指针，用好指针，成为每个 C++ 程序员的必修课。

3.9.1 指针就是表示内存地址的数据类型

指针，究其本质，是指一种表示内存地址的数据类型，它跟常用的表示整数的 int、表示浮点数的 float 一样。只是指针表示的是内存的地址，程序可以通过指针直接操作内存。为了更好地理解指针，先来看看内存空间的访问形式。

内存是跟程序打交道的最主要的硬件。一个程序最主要的过程就是通过读取内存中的数据来进行一定的运算，再将结果数据写回内存中。那么，C++ 程序是如何访问内存中的数据呢？在 C++ 程序中，有两种途径可以访问内存。一种是通过变量名间接访问。为了保存数据，通常会先定义保存数据的变量，再通过变量名间接地访问在内存中存放的数据，进行读取或者写入。另外一种方式是直接通过内存地址，也就是通过指针来访问内存中的数据。

在典型的 32 位计算机平台上，可以把内存空间看成是由很多个连续的小房间构成的，每

个房间就是一个小存储单元，大小是 1B，房间中住着数据。有的数据比较小，比如一个 char 类型的字符，它只需要一个房间就够了。而有的数据比较大，就需要占用好几个房间，比如一个 int 类型的整数，其大小是 4B，需要 4 个房间才可以安置。为了方便找到住在这些房间中的数据，房间就需要按照一定的规则编号，这个编号，就是通常所说的内存地址。这些编号是用一个 32 位的二进制数来编码的，比如 0x7AE4074B、0xFFFFFFFF 等，如图 3-6 所示。一旦知道某个数据的房间编号，就可以通过这个编号来对相应房间中的数据进行存取操作。C++中为了灵活地操作内存，特别内建了一种特殊的数据类型，以用来存放内存单元的地址，这就是指针。存放在指针中的内存地址可能是一个对象的地址、一个整数的地址，甚至是一个函数的地址。一般来说，如果指针变量所保存的是一个对象或者函数的地址，就说这个指针指向这个对象或者函数。

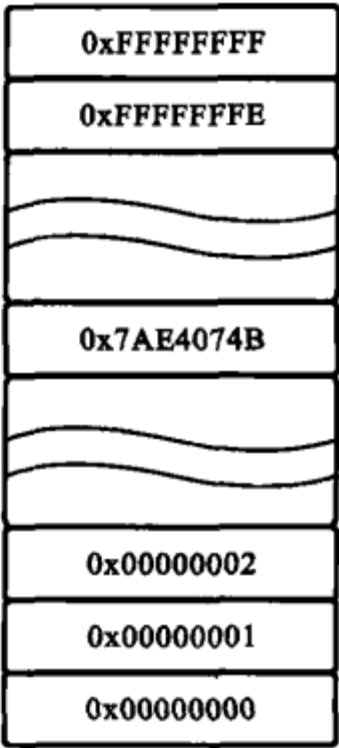


图 3-6 内存被划分为许多小的单元房间

3.9.2 指针变量的定义

既然指针是一种表示内存地址的数据类型，当然也可以作为变量类型，声明一个指针类型的变量。定义指针变量的语法格式如下：

数据类型* 变量名；

其中，“数据类型”表示指针所指向的数据是何种数据类型。这个数据类型可以是 int、string 和 double 等基本数据类型，也可以是自定义的结构体等复杂数据类型。通常，数据类型也称为指针的类型。把一个指向整型数据的指针称为整型指针。“*”表示声明的是一个指针类型的变量。“变量名”就是给这个指针指定的名字。例如：

```
// 一个 int 型指针，它可以指向一个 int 型数据
int* pnStudentNumber;
// 一个 double 型指针，它可以指向一个 double 型数据
double* pStudentHeight;
// 一个 Employee 型指针，它可以指向一个 Employee 结构体数据
Employee* pEmployee;
```

最佳实践：选择合适的定义指针变量的方式

实际上，下面两种定义指针变量的形式都是合乎 C++语法的：

```
int* pnStudentNumber;  
int *pnStudengNumber;
```

这两种形式都可以编译通过，并表示相同的语法含义。但是，这两种形式所反映的编程风格和对代码阅读者所强调的意义不同。

“int* pnStudengNumber”强调的是 pnStudentNumber 为一个指向整数的指针，这里，可以把 int* 看成为一种数据类型，而整个语句强调的是 pnStudentNumber 为这种数据类型的一个变量。

“int *pnStudengNumber”则是把 *pnStudentNumber 当成一个整体，表示它的数据类型是整型，而 pnStudengNumber 是指向这个整数的指针。

这两种形式没有对与错的区别，只有个人喜好的区别。推崇第一种形式，它把指针也当成是一种数据类型，更加清晰明了，可读性更强。

特别地，当在一条声明语句中声明多个指针变量时，可能会让人混淆，例如：

```
// 可能出错：p 是一个 int 类型的指针，而 p1 实际上是一个 int 类型的变量  
int* p, p1;  
// 清楚一些：*p 是一个整数，p 是指向这个整数的指针，p1 也是一个整数  
int *p, p1;
```

在实际开发中，有这样一条编码规范：“一条语句只完成一件事情”。按照这条规范，每条语句只定义一个变量，就可以很好地避免上述问题。

3.9.3 指针的赋值和使用

在得到一个指针变量之后，指针变量的值还是一个随机值。这个值可能是内存中无关紧要的数据，也可能是重要的数据或者程序代码，如果直接使用是很危险的，所以在使用指针之前，必须对其进行赋值，将其指向某个有意义的数据或代码。对指针变量进行赋值的语法格式如下：

指针变量 = 内存地址；

可以看到，对指针变量的赋值，实际上就是将这个指针指向某一内存地址，而这个内存地址上存放的就是这个指针想要指向的数据。通常用一个变量来保存数据，那么该如何方便地得到一个变量在内存中的地址呢？反过来，如果知道一个指针，又如何取出存放在其中的数据呢？为了解决这两个问题，C++ 提供了两个与内存地址相关的运算符——“&”和“*”。

1. “&”运算符

“&”称为取地址运算符。如果把它放在一个变量的前面，则可以得到该变量在内存中存放的地址。例如：

```
// 定义一个整型变量
int N = 703;
// 取得整型变量的地址并将其赋值给整型指针
int* pN = &N;
```

通过“&”运算符可以取得 N 这个整型变量的内存地址，然后将其赋值给指针 pN，也就是将指针 pN 指向 N 这个整数数据，如图 3-7 所示。

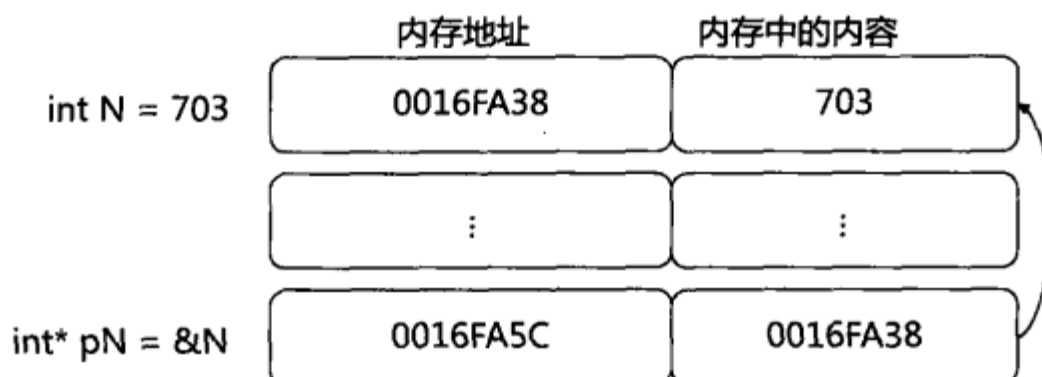


图 3-7 指针和指针所指向的数据

2. “*” 运算符

“*” 称为指针运算符。也称为解析运算符，它所执行的是跟“&”运算符相反的操作。如果把它放在一个指针变量的前面，就可以取得这个指针所指向的内容，无论这个内容是数据或者程序代码。例如：

```
// 通过指针修改它所指向的数据
*pN = 1983;
// 通过指针读取它所指向的数据
cout<<*pN<<endl;
```

通过“*”运算符可以取得 pN 这个指针所指向的数据变量 N，虽然“N”和“*pN”的形式不同，但是它们都代表内存中的同一份数据，都可以对这个数据进行读/写操作，并且是等效的。

特别地，如果一个指针指向的是一个结构体类型的变量，则可以直接使用“->”运算符访问这个结构体变量的成员变量。例如：

```
// 定义一个结构体变量
Employee Jiawei;
// 定义一个指针，并将其指向这个结构体变量
Employee* pJiawei = &Jiawei;
// 用“->”运算符，访问这个结构体变量的成员变量
pJiawei->m_strName = "Jiawei";
```

通过指针可以访问它所指向的数据，这使得在程序之间传递数据可以使用传递指向数据的指针来代替。因为指针只是这些数据在内存中的“门牌号”，所以传递指针要比传递真实

的数据快得多，而指针“四两拨千斤”的效果也正是如此体现的。

对于一个变量，可以使用取地址运算符“&”来得到它的地址，然后赋值给指针。同时，因为指针本身保存的就是内存地址，所以也可以使用一个指针来对另一个指针进行赋值，将两个指针指向同一个内容。例如：

```
// 定义一个整型变量
int a;
// 取地址运算符“&”得到变量 a 的内存地址并赋值给指针 pa
int* pa = &a;
// 使用一个已经存在的指针对另外一个指针赋值
int* pb = pa;
```

从上面的代码中发现，pa 和 pb 的值是相同的，都是变量 a 的地址，也就是说，两个指针可以指向相同的位置。



将语句编织成程序

见过 C++ 世界的“芸芸众生”之后，就知道如何使用各种类型的数据变量来描述现实世界中的各种事物了。使用数据变量，几乎可以写一个工资统计程序，代码如下：

```
// 工资统计程序
int _tmain(int argc, _TCHAR* argv[])
{
    // 保存所有工资的数组
    int nSalary[100];
    // 保存平均工资、最高工资和最低工资的变量
    float fAverageSalary;
    int nSalaryMax;
    int nSalaryMin;

    // .....对工资进行处理
    return 0;
}
```

在这段代码中，使用数组保存了所有员工的工资，用各种数据类型的变量保存了平均工资、最高工资和最低工资等，仿佛一个庞大的工资统计程序马上就要实现。但是，现在只知道如何用数据描述现实世界，对于如何处理数据以解决问题还一无所知。比如，我们不知道如何用加、减、乘、除计算平均工资；也不知道如何计算最高工资和最低工资。程序的两个目的——描述数据和处理数据，现在只完成了第一步，用变量描述现实世界中的数据；第二步就是要对数据进行处理，最终解决问题。

为了完成工资统计程序，下面来看看在 C++ 世界中如何对数据进行处理吧！

4.1 用运算符对数据进行运算

了解了 C++ 的基本数据类型，就可以用这些数据类型的变量来描述现实世界中常用的各种数据了。对数据处理最常见的就是对数据进行运算，以获得某个运算结果。例如，对两个数进行加法运算，可以得到它们的和；比较两个数的大小，可以得到它们的大小关系等。那么，在 C++ 中，如何对这些数据进行运算以得到最终想要的结果呢？换句话说，如何用 C++ 程序设计语言将我们头脑中对数据的处理表达出来？

4.1.1 用表达式表达设计意图

在计算数学题的时候，如果想知道两个数的和，总是先列出算式，然后再根据运算符的意义计算整个算式的值。在 C++ 中也是这样，如果想对数据进行处理，就要用表达式来描述运算的过程，再按照这个表达式所描述的运算过程对数据进行处理，最终获得整个表达式的结果。在 C++ 中，表达式由运算符、操作数和标点符号三部分组成，其作用是描述一个计算过程。表达式的核心是运算符和操作数。表达式中的常数、变量等数据量称为操作数，而连接这些操作数的是操作符。操作数是操作符的处理对象，而操作符则表达了对所连接的操作数的处理方式。比如，一个加法运算表达式“ $a + 5$ ”中，变量“ a ”和常数“ 5 ”是这个表达式的操作数，而连接这两个操作数的是加法运算符“ $+$ ”，表示将它所连接的两个操作数相加。例如：

```
// a、5 是操作数，=是运算符，它描述的计算过程是将 5 赋值给变量 a
a = 5;
// b、a、5 是操作数，=、+都是运算符
// 它描述的计算过程是计算变量 a 和常数 5 的和，然后将其赋值给变量 b
b = a + 5;
// a、b、c 是操作数，<、?、:是运算符，()是标点符号
// 它描述的计算过程是比较 a 是否小于 b，如果 a 小于 b，
// 则将 a 赋值给 c；反之，则将 b 赋值给 c
c = ( a < b ) ? a : b;
```

表达式中的运算符决定了整个表达式中操作数的个数。例如，加法运算需要两个操作数，一个是加数，另外一个是被加数。这种需要两个操作数的运算符称为二元运算符，如常见的加、减、乘、除运算符。另外一些运算符需要一个或者三个运算符，这种运算符分别称为一元运算符和三元运算符。C++ 提供了丰富的运算符，用于表达数据之间复杂的运算关系，如有表示加、减、乘、除的算术运算符，也有表示大小关系的关系运算符等。下面就来简单介绍这些常用的运算符及由它们连接而成的各种表达式。

4.1.2 算术运算符

在开发实践中，用得最多的就是算术运算符了。C++ 提供的基本算术运算符有以下几种。

- $+$ (加)：计算两个数的和。
- $-$ (减)：计算两个数的差。
- $*$ (乘)：计算两个数的积。
- $/$ (除)：计算两个数的商。
- $\%$ (取余)：计算两个数的余。

以上算术运算符的意义跟数学中相应运算符的意义是一致的。运用这些算术运算符可以很方便地表达对数据的算术运算。例如：

```
1 + 2;    // 对 1 和 2 这两个操作数进行加法运算，表达式的值为 3
4 * 5;    // 对 4 和 5 这两个操作数进行乘法运算，表达式的值为 20
10 % 7;   // 对 10 除以 7 取余，表达式的值为 3
```

另外，为了提高工作效率，C++还提供了针对整数的“++（自增）”和“--（自减）”两个运算符。它们都是一元运算符，用于完成对操作数的加 1 或者减 1 的运算。这两个运算符可以分别放在操作数的前面和后面，分别称为前置运算符和后置运算符。例如，最常见的循环语句：

```
// ++i 就是用自增运算符对 i 进行加 1 运算
for( int i = 0; i < 10; ++i )
{
    // ...
}
```

在这个循环语句中，就使用了自增运算符来完成对循环变量的处理。

这里需要特别注意的是，当这两种运算的结果要用来继续运算时，前置运算符和后置运算符的效果是不一样的。观察下面这段代码：

```
int a = 1;    // 声明整型变量 a，并给 a 赋初始值为 1
cout<<++a;    // 利用自增运算符对 a 加 1，输出为 2，这时 a 的值为 2
cout<<a++;    // 利用自增运算符对 a 加 1，输出还是 2，但是 a 的值为 3
```

第二条语句的输出为 2，这是因为当使用前置自增运算符时，a 首先进行自增运算，其数值变为 2，然后再输出 a 的值 2。但是大家一定会对第三句输出也为 2 感到奇怪，这是因为使用了后置自增运算符，输出语句首先要输出 a 的当前值 2，然后 a 再进行自增运算，其值变为 3。

既然前置自增运算符和后置自增运算符容易让人产生误解，那么，在软件开发中有这样一条经验：使用前置自增运算符代替后置自增运算符。这条经验可以带来如下好处。

1. 前置运算符的效率优于后置运算符

在 C++ 底层，后置运算符是通过前置运算符实现的，实质上，使用后置运算符最终使用的还是前置运算符，并且增加了额外的转换消耗。所以，使用前置运算符可以提高代码的执行效率。

2. 前置运算符不易使人产生误解

后置运算符有时会让人产生误解。例如：

```
int a = 1;
int b = 0;
b = 1 + a++; // 变量 b 的值到底是 2 还是 3 呢？
```


这段代码执行完成后，b 的值是 2，而不是 3。这是因为在进行加法运算时，使用的是 a 的当前值 1，而不是其自增后的值 2。如果使用前置运算符，则不会产生这样的误解。

4.1.3 赋值操作符

有了算术运算符，就可以得到各个数据的算术运算结果。但是有了运算结果，还需要把结果保存下来以备后用。赋值操作符就是用来保存运算结果的。在 C++ 中，最简单的赋值操作符为“=”。赋值操作符是一个二元运算符，其作用就是将等号右边表达式的值赋给等号左边的对象。例如：

```
int a, b, c;
a = 1;           // 将 1 赋值给变量 a，a 的值变成 1
a = b = c = 1;   // 连续赋值，a、b、c 的值都变成 1
a = a + 1;        // a + 1 表达式的值为 2，将其赋值给 a，a 的值变成 2
```

另外，C++ 还提供了几个复合运算符，包括算术运算符与赋值操作符复合，例如 +=、-=、*=、/=、%=；位运算符与赋值操作符复合，例如 <<=、>>=、&=、^=、|=。

复合运算符同样是二元运算符，它们将运算符两边的操作数通过复合运算符中的算术运算符或者位运算符进行运算，然后将结果赋值给左边的操作数，从而实现计算和赋值的复合。例如：

```
a += 1;          // 等价于表达式 a = a + 1;
a *= b + 1;       // 等价于表达式 a = a * (b + 1);
```

4.1.4 关系运算符

“关系”还可以运算？是的。所谓关系运算，就是将两个操作数进行比较，得出一个结论。这个结论就是这两个操作数之间的关系。很费解是不是？不要紧，看下面这个例子就明白了。比如你正在找女朋友，现在有两个候选对象，该如何选择呢？这时你可以对这两个候选对象进行关系运算：小芳跟小花比较，小芳更漂亮一些。

这里，“小芳跟小花比较”是对小芳和小花两个对象进行关系运算，得出的关系结论是“小芳更漂亮一些”。当然，在 C++ 中不会去比较两个操作数谁更漂亮，但是会比较两个操作数的大小，等等。为了进行关系运算，C++ 中提供的关系运算符包括以下几种：>（大于），>=（大于或等于），<（小于），<=（小于或等于），==（等于），!=（不等于）。

关系运算符是二元运算符，其作用是将两个操作数进行关系运算，比较两个操作数的大小或者是否相等，其运算结果类型为 bool 类型。如果两个操作数的关系符合运算符，则表达式的结果为 true，反之则为 false。例如：

```

int a = 5;
int b = 3;
// a 的值大于 b 的值, 符合运算符 ">" 的意义, 其运算结果为 true
a > b;
// a 的值不等于 b 的值, 不符合运算符 "==" 的意义, 其运算结果为 false
a == b;
// 将分数 nScore 跟及格分数 60 进行关系运算
// 如果 nScore 大于等于 60, 则表示这个分数是及格分数,
// bPass 的值为 true, 否则为 false
bool bPass = nScore >= 60;

```

关系运算符常用来判断某种条件是否成立。例如, 要表达“未满 18 岁不准进入”的含义, 可以将代码写成:

```

if ( nAge < 18 )      // 利用关系运算符, 判断年龄 nAge 是否小于 18
{
    // 不准进入
}

```

最佳实践: 不要使用 “==” 比较两个浮点数是否相等

编译并运行下面这段代码:

```

// 两个“相等”的浮点数
float x = 0.5;
double y = 0.5;

// 使用 “==” 判断两个浮点数的运算结果是否相等
if( cos(x) == cos(y) )
    cout<<"x 等于 y"<<endl;
else
    cout<<"x 不等于 y"<<endl;

```

如果输出结果是“x 不等于 y”, 就不要吃惊。这是因为在 C++ 中, 比较两个浮点数是否相等是不保险的。即使在表面看来 x 等于 y, 但是当使用 “==” 关系运算符比较这两个浮点数是否相等时, 因为浮点数精度的差异, 其比较结果可能是 x 不等于 y。使用 “==” 关系运算符比较两个浮点数的结果, 取决于计算机硬件和编译器优化设置。这段代码在某台计算机上输出的结果可能是“x 等于 y”, 但是在另外一台计算机上输出的结果却又可能是“x 不等于 y”。所以, 为了保证代码行为的一致性, 不要在代码中使用关系运算符 “==” 比较两个浮点数是否相等。

如果确实需要在代码中比较两个浮点数是否相等又该怎么办呢? 最简单的方法就是设定一个允许的误差值, 当两个浮点数相减的结果在这个误差范围内时, 就认为两个浮点数相等, 反之则认为两个浮点数不相等。例如, 上面的代码可以修改为下面的样子, 以保证代码行为的一致。

```

float x = 0.5;
double y = 0.5;

// 设定允许的误差值
const double fEpsilon = 0.00001;

```

```
// 判断两个浮点数相减的结果是否在允许的误差范围内
// 如果相减的结果在误差范围内, 则认为两个浮点数相等
if( fabs( cos(x) - cos(y) ) < fEpsilon)
    cout<<"x 等于 y"<<endl;
else
    cout<<"x 不等于 y"<<endl;
```

经过这样的改写, 程序代码在任何计算机上都可以得出正确的结果。

针对关系运算用做条件判断的情况, C++还专门提供了一种运算符用于简化关系运算和条件判断, 这就是条件运算符。

条件运算符是 C++提供的唯一一个三元运算符, 它需要三个操作数, 其语法格式如下:

条件判断表达式 ? true 表达式 : false 表达式

其中, 条件判断表达式是在整个表达式中进行的条件关系判断, 其结果是 bool 类型, 或者能够转换成 bool 类型。条件运算符首先计算条件判断表达式的值, 如果其值为 true, 则接着计算 true 表达式并将其结果作为最终结果; 反之, 则计算 false 表达式的值并将其结果作为最终结果。这样就可以根据条件判断表达式的值实现简单的条件选择功能。例如:

```
int a = 1;
int b = 2;
// 因为 a > b 这个条件判断表达式的值为 false, 而整个表达式的值为 2, 所以 max 的值为 2
int max = ( a > b ) ? 1 : 2;
```

在实际开发中, 条件运算符最常见的一个应用是选出两个数中较大或者较小的一个, 例如:

```
int max = ( a > b ) ? a : b; // 求 a 和 b 之间的较大值
```

这个表达式是先比较 a 和 b 的大小, 如果 a 大于 b, 则将 a 作为表达式的最终结果赋值给 max。如果 b 大于 a, 则将 b 作为表达式的最终结果赋值给 max。这样, 整个表达式的值始终是 a 和 b 中较大的一个, 可以很方便地求出 a 和 b 中较大的值。

4.1.5 逻辑运算符

处理复杂事务时, 除了对数据进行算术运算外, 常常还需要根据多种条件进行判断, 根据不同条件采取不同的处理方式。当有多个条件同时存在时, 就涉及对多个条件的逻辑运算, 得出一个最终的结果。

关系运算的结果是 true 或者 false。有了关系运算的结果, 还需要使用逻辑运算符将这些结果连接起来, 以构成更加复杂的逻辑表达式, 完成更加复杂的逻辑判断。C++提供的逻辑运算符包括以下几种。

- !(非): 计算操作数的逻辑非。

- &&(与): 计算两个操作数的逻辑与。
- ||(或): 计算两个操作数的逻辑非。

其中, “!” 是一元运算符, 只能放在操作数的前面, 例如:

```
bool bFlag = true;    // 声明一个 bool 类型的变量并赋值为 true
!bFlag;               // 对变量 bFlag 进行取非运算, 整个表达式的结果为 false
```

“!” 运算符的作用是对操作数取反。这里它的操作数 bFlag 的值为 true, 取反之后整个表达式 “!bFlag” 的值为 false。

“&&” 和 “||” 都是二元运算符。“&&” 的作用是求两个操作数的逻辑与, 只有当两个操作数的值都为 true 时, 整个表达式的结果才为 true。“||” 的作用是求两个操作数的逻辑或, 只要两个操作数中有一个为 true, 整个表达式的结果就为 true。例如:

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
// 关系表达式 a < b 和 c < d 的值都是 true, 所以 && 运算的结果为 true
( a < b ) && ( c < d );
// 关系表达式 a > b 的值是 false, c < d 的值是 true,
// 但是我们使用的是 “||” 运算符, 只要一个操作数为 true, 整个运算的结果仍然为 true
( a > b ) || ( c < d );
```

逻辑运算符常常和关系运算符组合起来用来表达比较复杂的条件判断。例如, 要表达 “未满 18 岁或者没有带够钱的不准许进入” 的含义, 则可以将代码写成:

```
// nAge 的值小于 18 或者 bEnoughMoney 的值为 false 时, 不准许进入
if ( nAge < 18 || (!bEnoughMoney) )
{
    // 不准许进入
}
```

4.1.6 运算符之间的优先顺序

当处理复杂的事物时, 在同一个表达式中, 有时可能会存在多个运算符。下面还是继续来看刚才的例子:

```
int nAge = 15;
bool bEnoughMoney = false;
if ( nAge - 2 < 16 || !bEnoughMoney )
{
    // 不准许进入
}
```

在这个表达式中, 有算术运算符、关系运算符和逻辑运算符。那么, 这么多运算符, 该从哪一个运算开始呢? 这个表达式的最终结果又是什么呢? 这时就需要弄清楚各个运算符之

间的优先顺序，知道哪个运算符，可以优先计算；哪个运算符，最后才有计算的机会。只有按照正确的计算顺序，才能得出表达式正确的结果。

在 C++ 中，各个运算符的优先级如表 4-1 所示。

表 4-1 运算符的优先级

级别	运 算 符	说 明
1	()	括号就是所有运算符中的领导，具有最高的优先级
2	!、+(正号)、-(负号)、++、--	+、-指的是改变数值正负属性的符号
3	*, /, %	乘、除、取余运算
4	+, -	加、减运算
5	>, >=, <, <=, ==, !=	关系运算符
6	&&	逻辑与
7		逻辑或
8	=, +=, *=, /=, %=	赋值操作符

在 C++ 中计算表达式的结果时，总是优先计算优先级高的运算符；同一级运算符，则按照从左到右的顺序进行计算。了解了各运算符的优先级，就可以计算这个复杂表达式的结果了。在这个表达式中优先级最高的运算符是“!”，所以它优先得到运算，形成这样的中间结果：

```
nAge - 2 < 16 || true
```

在这个中间结果表达式中，优先级最高的是算术运算符“-”，计算这个算术运算符，得到一个中间结果：

```
13 < 16 || true
```

经过前面两步的运算，整个表达式就清晰多了。现在可以开始计算优先级比较高的关系运算符“<”，得到的中间结果是：

```
true || true
```

现在，事情一目了然了，表达式的最终结果是 true。

从这个过程中可以发现，过于复杂的表达式计算起来非常麻烦。虽然表达式是由程序进行计算的，但是表达式是由程序员设计，并且也要提供给他人的阅读的。过于复杂的表达式容易出错。所以我们应当减少表达式中多个运算符的混合使用，尽量保持表达式的短小精悍。

如果确实需要使用比较复杂的表达式，比如多个条件的复合判断等，则可以使用“()”人为地标识或者改变表达式中运算的优先关系，来反映所设计的整个表达式的计算顺序。比如，可以用括号改写上面的表达式，让其表达的意义更加清晰，更加接近思考计算过程：

```
if ( ( (nAge - 2) < 16 ) || (!bEnoughMoney) )
{
    // 不准许进入
}
```

使用括号后，整个表达式的运算过程就比较简单了：首先计算年龄，判断年龄是否小于16，然后判断是否有足够多的钱，最后将这两个条件进行逻辑运算，得到了最终的结果。

4.1.7 将表达式组织成语句

学习 C++ 编程实际上也就是学习如何用这门程序设计语言说话。在前面的章节中学习了表达式，这只是相当于学习了这门语言的只言片语，只知道“一个变量 a”和“三加二……”：

```
// 只言片语的表达式
a;
3 + 2;
```

程序可以执行这些表达式，但这些表达式并不改变程序的运行逻辑，没有任何实际的意义。在 C++ 中，把一些表达零散意义的表达式组合起来，完成某个相对完整的功能后，再加一个分号表示结束，就组成了一条语句。例如，把这两个表达式通过赋值操作符组合起来，就形成了一条完整的赋值语句：

```
// 赋值语句
a = 3 + 2;
```

这样就清楚多了，它表达了一个完整的意义：计算 3 和 2 的和，然后赋值给变量 a。

在 C++ 中，语句和表达式并没有严格的区分。一个表达式加上一个分号就可以直接形成语句。语句强调它所完成的功能，而表达式关注它所描述的运算和最终的结果。我们现在已经学过两种常用的语句类型，其一为变量定义语句，其二为赋值语句。

下面用变量定义语句来完成变量的定义。

```
// 变量定义语句
int a ,b, c;
```

赋值语句则是为指定变量赋予某个指定值。

```
// 赋值语句
a = 20;
c = 2 * (a + b);
```

当连续的多个语句属于同一控制范围时，可以用一对花括号“{}”将这些语句括起：

```
{
    int a = 100;
    bool b = a > 20;
    // 更多代码...
}
```

花括号内的内容，称为复合语句。

就像我们说话可以很复杂一样，C++中的语句当然也不会这么简单，接下来将学习 C++ 中更加复杂的语句。

4.2 条件选择语句

“爱我还是她？”如果你被问到这个问题，你会如何回答？是的，你会根据不同的条件选择不同的回答。如果问你这个问题的是一位天仙，那回答当然是：“爱你！”但是如果问你这个问题的是“一只恐龙”，那就另当别论了。

这就是条件选择，我们总会根据不同的条件做出不同的回答，人生就是由这样的一个个选择构成的。程序如人生，人生如程序。自然，在 C++ 程序中也少不了条件选择。

4.2.1 if 语句

我们总是用“如果……，就……”来表达条件选择，C++也向我们学习，提供了关键字 if 来实现选择结构。if 语句的语法格式如下：

```
if ( 条件表达式 )
{
    语句 1;
}
else
{
    语句 2;
}
```

在条件选择语句中，首先计算条件表达式的值，然后根据表达式进行判断。如果表达式的值为 true，则执行语句 1；否则执行语句 2。通过使用条件选择语句，可根据条件表达式的不同值而改变程序执行的流程，可以在语句 1 和语句 2 中实现不同的功能。if 语句的执行过程如图 4-1 所示。

现在，就可以用 if 语句来解决“爱我还是她”这个令人烦恼的问题了。

```
// 得到漂亮指数
int nBeautyIndex = GetBeautyIndex();
// 以漂亮指数作为条件进行判断选择
if( nBeautyIndex > 85 )
{
    // 如果“漂亮指数大于 85”条件成立，选择“爱你”
    cout<<"爱你！"<<endl;
}
else
```

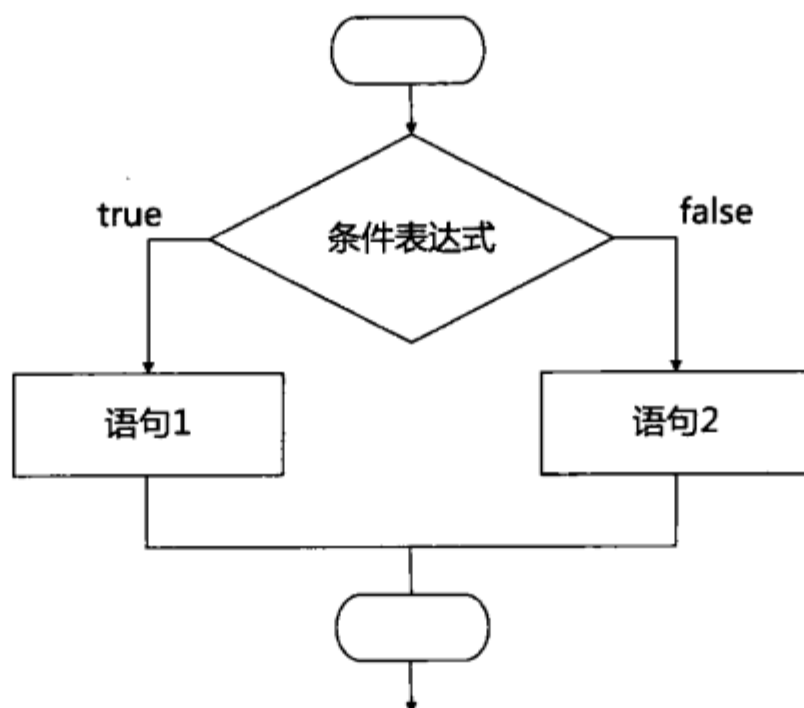



图 4-1 条件选择结构的执行流程

```
{  
    // 如果“漂亮指数大于 85”条件不成立，选择“爱她”  
    cout<<"爱她!"<<endl;  
}
```

在这里，首先得到漂亮指数，然后将漂亮指数跟一个标准进行比较，再将比较的结果作为条件进行判断。如果漂亮指数大于 85，则比较结果为 true，执行条件选择结构中的语句 1，输出答案为“爱你！”；如果条件不能满足，则执行语句 2，输出答案为“爱她！”。用程序解决问题就是这么简单。从现在开始，谁也别爱了，爱上 C++ 吧。

if 语句的形式虽然简单，但是在其使用上有如下几个需要注意的地方。

1. if 语句中的 else 部分可以省略

很多时候，我们只关心条件为 true 的情况，即省略掉 if 语句中的 else 部分，只保留 if 判断条件和相应的语句，例如：

```
// 如果分数大于 80，则输出“Good”  
if( nScore > 80 )  
{  
    cout<<"Good!";  
}
```

这个条件选择结构表示，如果 nScore 的值大于 80，就输出“Good！”以示嘉奖。对于 nScore 小于或等于 80 的情形，我们不关心，也不处理。

2. if 语句可以实现嵌套

if 语句可以嵌套在 if 语句中，表示在某个前提条件下做进一步的条件判断，从而实现更加复杂的选择。例如，要比较输入的 v1 和 v2 的大小关系，可以使用如下代码。

```

cout << "请输入两个整数:" << endl;
int v1, v2;
// 获取用户输入的数字
cin >> v1 >> v2;
if( v1 != v2 )    // 判断 v1 和 v2 是否相等, 如果不相等, 则继续判断大小
{
    // 第二级 if 语句
    // 如果不相等, 则继续判断 v1 是否大于 v2
    if( v1 > v2 )
    {
        cout<<" v1 > v2 "<<endl;
    }
    else
    {
        cout<<" v1 < v2 "<<endl;
    }
}
else    // v1 和 v2 相等
{
    cout<<" v1 = v2 "<<endl;
}

```

在这段代码中, 首先判断 v1 和 v2 是否相等。如果不相等, 则继续判断 v1 是否大于 v2, 如果还是不满足, 则直接输出结果。通过两级判断, 实现了 v1 和 v2 大小的比较。

3. if 语句可以并列

如果同一级的条件有多种情况, 就可以使用并列的条件选择语句来实现。并列条件选择语句的语法格式如下:

```

if ( 条件表达式 1 )
{
    语句 1;
}
else if ( 条件表达式 2 )
{
    语句 2;
}
// ...
else if ( 条件表达式 n )
{
    语句 n;
}
else
{
    语句 n+1;
}

```

当程序开始执行一条并列条件选择语句时, 会从上到下逐个计算条件表达式的值, 并判断是否为 true, 如果条件表达式的值为 true, 则进入相应的分支, 执行其中的语句, 完成整个并列条件选择语句的执行。这里需要特别注意, 并列条件选择语句只会执行其中的某一个

分支，如果多个条件表达式都为 true，则只会执行第一个分支。例如：

```
// 并列条件选择语句的执行
int a = 5;
if( a > 1 ) // 条件表达式为 true，执行此分支
{
    // ...
}
// 条件表达式也为 true，但是第一个分支已经执行，
// 所以虽然此分支的条件表达式也为 true，但是不会执行
else if( a > 2 )
{
    // ...
}
else
{
    // ...
}
```

当使用并列条件选择语句时，应尽量避免条件范围的重复覆盖，不要让多个条件表达式同时为 true，这样可能会造成程序逻辑上的混乱。之前比较 v1 和 v2 大小的例子中，会出现三种不同的情形，即 $v1 > v2$ 、 $v1 < v2$ 和 $v1 = v2$ ，所以可以将其代码改为采用并列条件选择语句的形式。

```
int v1, v2;
// 获取用户输入的数字
cin >> v1 >> v2;
if( v1 < v2 ) // 判断 v1 是否小于 v2
{
    cout<<" v1 < v2 "<<endl;
}
else if( v1 > v2 ) // 判断 v1 是否大于 v2
{
    cout<<" v1 > v2 "<<endl;
}
else // 如果 v1 既不大于 v2，又不小于 v2，则 v1 和 v2 必然相等
{
    cout<<" v1 = v2 "<<endl;
}
```

通过 if 语句的并列，我们对各种条件判断的可能结果都进行了相应的处理，没有重复，也没有遗漏。

4.2.2 并列选择的 switch 语句

并列条件选择语句显然是用于需要进行多次判断才能做出选择的情况。为了简化这种多次并列的条件判断，C++ 提供了专门的 switch 语句以代替复杂的并列条件选择语句。switch 语句的语法格式如下：

```

switch( 条件表达式 )
{
case 常量表达式 1:
{
    语句 1;
}
break;
case 常量表达式 2:
{
    语句 2;
}
break;
//...
case 常量表达式 n:
{
    语句 n;
}
break;
default:
{
    语句;
}
}

```

其中，条件表达式就是要进行判断的条件。switch 语句首先计算条件表达式的值，这个表达式的值只能是整型或字符型。完成这个表达式的计算之后，程序开始在各个“case”分支中从上到下逐个匹配，查找哪个常量值和这个表达式的值相等。如果找到相等的常量表达式，则以此为入口开始往下顺序执行 case 分支中的语句，直到遇到 break 关键字，完成整个 switch 语句的执行。如果查找所有 case 分支都没有找到相等的常量表达式，则进入表示默认情况的 default 分支开始执行，最终完成整个 switch 语句。default 关键字是可选的，如果没有 default 关键字，程序又找不到匹配的 case 分支时，则直接结束 switch 条件选择语句的执行，如图 4-2 所示。

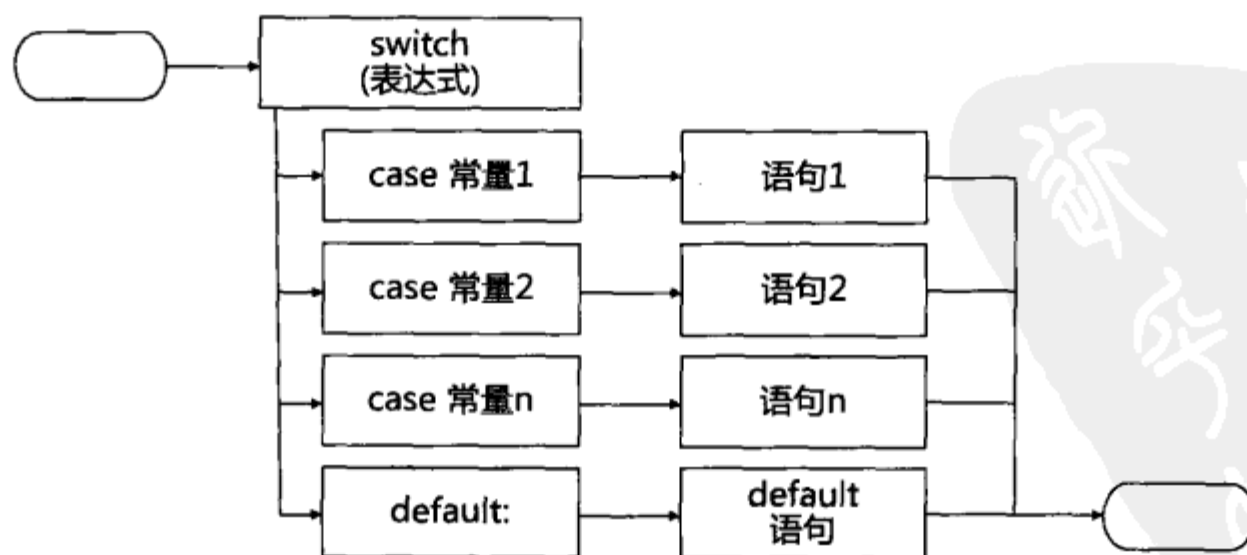


图 4-2 switch 语句的执行流程

为了更好地理解 switch 选择结构，我们来看一个实际的例子：调查大家是通过什么途径得知本书的。整个调查程序提供一个列表供用户选择，然后根据用户的选择输出用户得知本书的途径。

```
// 读者调查程序
int _tmain(int argc, _TCHAR* argv[])
{
    // 显示提示信息，提示用户进行选择
    cout << "感谢您购买本书！\n 请输入序号，选择您从何种途径得知本书。" << endl;
    cout << "1、通过搜索引擎" << endl;
    cout << "2、通过朋友介绍" << endl;
    cout << "3、通过书店" << endl;
    cout << "4、通过其他途径" << endl;

    // 接收用户输入的选项
    int nWay;
    cin >> nWay;

    // 使用 switch 选择结构，比对用户输入的选项，输出相应的信息
    switch (nWay)
    {
        case 1 :
            cout << "您是通过搜索引擎得知本书的。" << endl;
            break;
        case 2 :
            cout << "您是通过朋友介绍得知本书的。" << endl;
            break;
        case 3 :
            cout << "您是通过书店得知本书的。" << endl;
            break;
        case 4 :
            cout << "您是通过其他途径得知本书的。" << endl;
            break;
        default :
            cout << "错误的选择！请输入数字 1 ~ 4 做出选择。" << endl;
    }

    return 0;
}
```

这个读者调查程序首先输出提示信息，提示用户输入选项；然后接收用户输入的选项，并保存到变量 nWay 中；获得用户输入选项后，就开始执行 switch 语句，逐个将用户输入的 nWay 跟各分支进行比较，判断用户输入的序号，最后根据用户的选择输出相应的信息。

从以上代码中注意到，在每个 case 分支的末尾都添加了关键字 break。break 可以让程序跳出当前所在的 switch 条件选择语句，继续执行后面的语句。在本例中，如果执行遇到 break 关键字，则会跳出 switch 语句，直接执行“return 0;”。为什么这里需要 break 关键字呢？因为虽然 switch 语句在形式上跟并列的条件选择语句相似，但是其执行过程不同。在并列的条

件选择语句中，程序只会执行某个符合条件的分支，在 switch 语句中，程序进入某个分支语句后，会执行其后的所有分支语句，直到遇到 break 关键字。如果想要输出序号所对应的信息，需要在程序执行这个分支后就结束 switch 语句的执行。如果没有 break 关键字，就会执行后面的所有分支语句，输出其后所有的信息。例如，在使用 break 关键字的情况下，如果用户输入的是 3，则输出：

您是通过书店得知本书的。

如果没有 break 关键字，则会输出：

您是通过书店得知本书的。

您是通过其他途径得知本书的。

错误的选择！请输入数字 1 ~ 4 做出选择。

我们只是想要某个分支的输出结果，但是 switch 语句却输出了其后所有的信息，switch 语句做了自己不该做的事情，这显然不是我们想要的结果。所以在以后的开发实践中，一定要在 switch 分支语句结束的地方加上 break 关键字，结束整个 switch 语句的执行。

当然，凡事都有例外情况。当某些分支条件有共同的功能需要完成时，也可以去掉分支语句中的 break 语句，共用实现相同功能的代码，例如，KFC 的点餐程序。

```
// KFC 点餐程序
int _tmain(int argc, _TCHAR* argv[])
{
    // 显示提示信息，提示用户进行选择
    cout << "欢迎点餐，请选择您要的套餐" << endl;
    cout << "1. 鸡翅套餐" << endl;
    cout << "2. 可乐套餐" << endl;

    // 接收用户输入的选项
    int nOrder;
    cin >> nOrder;

    // 使用 switch 选择结构，比对用户输入的选项，输出相应的信息
    switch ( nOrder )
    {
    case 1 :
        cout << "一个鸡翅" << endl;
    case 2 :
        cout << "一杯可乐 + 一包薯条" << endl;
    default :
        cout << "餐巾纸" << endl;
        cout << "您的餐齐了" << endl;
    }

    return 0;
}
```

这里，我们去掉了 switch 语句中的 break 关键字，让各个 case 分支能够共用相同的部分。当选择鸡翅套餐进入第一个 case 分支语句时，程序会从上往下执行所有 case 分支语句，得到这样的输出：

```
一个鸡翅  
一杯可乐 + 一包薯条  
餐巾纸  
您的餐齐了
```

当选择可乐套餐时，就只有可乐加薯条了。这里，鸡翅套餐有自己独特的内容，同时又涵盖了可乐套餐的所有内容。另外，default 语句提供了所有套餐共有的内容。省掉 break 关键字，就可以共用各个分支的内容。break 关键字的灵活运用，让 KFC 繁忙的点餐变得很简单。

在使用 switch 条件选择结构时，还需要注意以下几个问题。

(1) switch 后的表达式是整型，或者是能够转换为整型的其他类型，比如字符型或者枚举类型。

(2) 因为要跟 switch 后的表达式进行比较，所以 case 之后必须是一个常量表达式。它可以是常量数字，如上面例子中表示选项的常量数字 1 和 2，也可以是常量计算表达式，但不能是变量或带有变量的表达式。

(3) 各个常量表达式的值不能相同，即不能出现两个相同条件的 case 分支。

4.3 循环控制语句

每天看到太阳从东边升起，西边落下；公共汽车总是隔一段时间就发出一辆；我们每天都是上班、工作、下班回家。这些现象都是现实生活中循环反复出现的。这种循环反复表现在程序中就是循环控制语句。

C++ 中提供了三种循环控制语句，以方便我们描述现实世界中的这种循环反复现象。

4.3.1 while 循环

在自然语言中，有这样一种表达循环反复的句式，如“只要……，就一直……”

在现实生活中，有很多这样的例子：只要给我加工资，我就一直好好干；只要天还没有黑，你就得一直给我干活。这些循环反复的情形都可以用 while 循环来表达。在 C++ 中，while 循环结构的语法格式如下：


```
while( 条件表达式 )
{
    循环体语句;
}
```

其中，条件表达式就是这个循环是否继续进行的条件，而循环体语句就是这个循环所做的事情。这样，while 循环控制语句就跟自然语言的句式很好地对应起来了。while 语句首先会判断条件表达式的值，如果表达式的值为 true，则执行循环体语句；如果为 false，则结束 while 语句。当循环语句执行完一次时，会再次判断表达式的值，根据表达式的值决定是否要进行下一次循环，如此不断循环，直到表达式的值为 false，循环结束。可以把 while 理解成自然语言的“只要”的意思，只要条件成立，就不断执行循环体语句。当条件不再满足时，就结束 while 循环控制语句。while 循环控制语句的执行流程如图 4-3 所示。

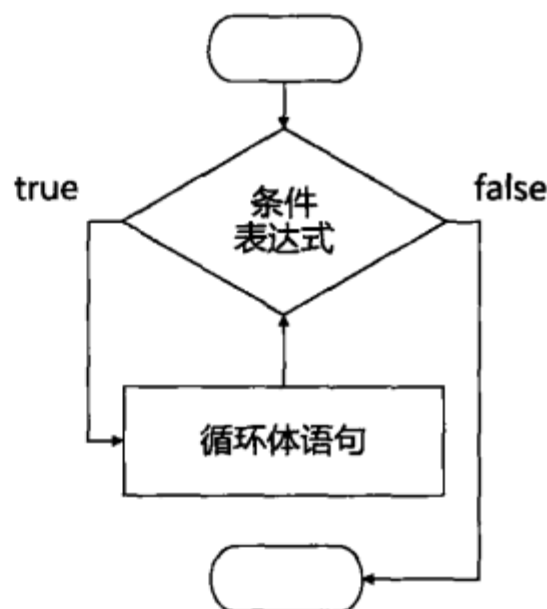


图 4-3 while 循环的执行流程

下面来看一个实际的例子，看 while 语句如何任劳任怨地为我们反复做一件事情。在生活中，精打细算的妈妈总是喜欢统计每月的收支情况，常常看到她们拿着计算器不停地按呀按，手指都按疼了。现在有了 while 循环，这些烦琐的事情就交给它去完成好了。

```
// 每月收支统计程序
int _tmain(int argc, _TCHAR* argv[])
{
    cout << "====每月收支统计程序====" << endl;
    cout << "请输入你本月的收入(正数)和支出(负数)，0 表示输入结束。" << endl;

    int nInput = 1;
    int nTotal = 0;
    // 判断用户输入
    // 如果用户输入不为 0，则继续执行循环
    // 如果用户输入为 0，则结束循环
    while( 0 != nInput )
    {
        cout << "请输入你的收入或支出：";
        cin>>nInput;
        // 对收支进行统计
        nTotal += nInput;
    }

    // 输出统计结果
    cout << "你本月的结余是：" << nTotal << endl;
}
```

```
    return 0;
}
```

这个收支统计程序首先会提示我们输入相应的收入和支出数据，在完成一次输入后，程序会将输入的数据累加到总量 `nTotal` 中，然后开始下一次循环，再次接收输入并进行累积，如此反复。直到最后输入 0，循环的条件不再满足而结束整个循环。最后，程序会给出本月的结余。整个过程我们只需要输入数据，程序会循环接收这些数据并进行统计，这可比按计算器省事多了。赶快把这个程序献给老妈，说不定会得到一番夸奖呢。

也别光顾着乐，还是来看看这个程序到底是如何工作的。在这段程序中，首先定义了两个数用来接收用户输入和保存计算结果，然后进入 `while` 循环语句。`while` 语句首先要对 `nInput` 的值进行判断，因为它有给定的初始值 1，当条件表达式的计算结果为 `true` 时，可以进入循环体内执行。循环体接收用户的输入并对收支进行统计。循环体执行结束后，`while` 语句再次判断 `nInput` 的值，如果输入的值不是 0，则继续执行；如果输入表示统计结束的数字 0，则 `while` 循环的条件不再满足，从而完成整个循环的执行。这就是整个 `while` 循环的执行过程，大家也可以用调试的方法，单步运行这个程序的整个执行过程，可以对整个循环的执行过程有一个更加清晰的认识。

4.3.2 do...while 循环

在以上 `while` 循环的例子中，我们注意到，`nInput` 需要给定初始值才可以完成整个循环。很多情况下，`while` 循环的条件没有合适的初始值，那么有没有办法可以改进上面的设计呢？有，C++ 提供了 `while` 循环的孪生兄弟——`do...while` 循环来解决这个问题。在 C++ 中，`do...while` 循环控制语句的语法格式如下：

```
do
{
    循环体语句;
}
while ( 条件表达式 );
```

虽然是孪生兄弟，但是 `do...while` 循环语句跟 `while` 循环语句不仅在形式上有差别，一个条件表达式在前，一个条件表达式在后，而且在执行顺序上两者也有差异。`do...while` 循环语句首先会执行一次循环体语句，然后再判断条件表达式的值。如果条件表达式的值为 `true`，则继续执行循环体语句；如果条件表达式的值为 `false`，则结束整个循环。`do...while` 循环语句的执行流程如图 4-4 所示。

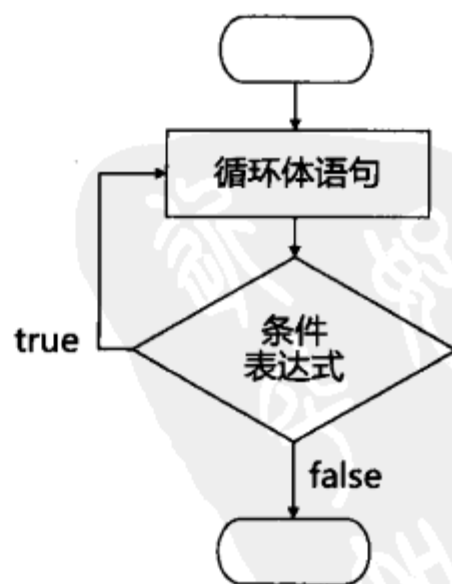


图 4-4 do...while 循环结构的执行流程

do...while 循环是 while 循环的一个变化形式,它们之间最本质的区别就是 do...while 循环要首先执行一次循环体语句,然后再判断表达式的值,而 while 循环会首先判断表达式的值,然后再决定是否执行循环体语句。这样就使得 do...while 循环可以应用在那些无论什么条件循环都至少执行一次的情况。同时,do...while 循环可以省掉很多进入循环前的初始化工作,在循环体中对控制循环的变量进行初始化。例如,以上例子使用 do...while 循环可以改写为:

```
// 注意, nInput 可以不初始化或者初始化为 0
int nInput = 0;
int nTotal = 0;
do
{
    cout << "请输入你的收入或支出: ";
    cin>>nInput;
    nTotal += nInput;
}while( 0 != nInput );
```

使用 do...while 循环结构改写后, nInput 无须给定初始值,可以直接由用户输入赋值,然后再进行条件判断。这里我们看到了 do...while 循环结构的优势:可以在第一次循环体语句的执行中改变表达式的值。在某些循环至少要完成一次的情况下,使用 do...while 循环结构更加合适。

4.3.3 for 循环

虽然 while 循环和 do...while 循环能够满足大多数情况下对循环控制的需要,但是 C++ 还是提供了另外一种更加强大的循环控制语句——for 循环。

for 循环控制结构是 C++ 中使用频率最高的,它通常用来表达在某个范围内的循环。在 C++ 中, for 循环控制结构的语法格式如下:

```
for( 初始化语句; 条件表达式; 更改语句 )
{
    循环体语句;
}
```

for 循环控制结构中的三个表达式很好地反映了循环控制结构的三个基本要素。

1. 初始化语句

在执行循环控制语句之前,一般都需要进行一定的初始化工作,比如确定循环的范围,或者给控制循环的变量一个合适的初始值。在前面 while 循环的例子中,给 nInput 赋值为 1 就是一个初始化工作。在 for 循环中,初始化语句就是将初始化工作提取出来集中放置在初始化语句中。

2. 条件表达式

任何循环都必须有结束循环的机会,否则就是一个死循环。条件表达式就是用来根据循

环控制变量的值进行判断，给循环一个结束的机会。例如，在前面的例子中，会根据 `nInput` 的值进行判断，如果 `nInput` 等于 0 就结束循环。

在使用条件表达式时务必小心，要确保条件表达式在循环结束时其值为 `false`。当循环无论怎么执行，条件表达式的值始终为 `true` 时，就会形成死循环。例如：

```
// 死循环
// 不停地说“I LOVE YOU”，不是疯子就是傻子
while ( true )
{
    cout << "I LOVE YOU!" <<endl;
}
```

无论循环怎么执行，这里的条件表达式的值始终是 `true`，所以循环就没法结束，从而构成一个死循环。应该尽量避免死循环的出现。

3. 更改语句

在各种循环结构中，总是有一个循环控制变量用来构成循环是否继续执行的条件。例如前面例子中的 `nInput` 就是一个循环控制变量，可以用它的值来判断是否需要进行下一次循环。既然是表示循环的条件，那就需要在循环中对这个变量进行修改，以反映循环的执行情况，根据执行情况决定循环是否继续进行。例如，将用户输入的值赋给 `nInput`，就是对循环控制变量的修改。`for` 循环是将循环控制变量的修改独立出来放到了更改语句中来进行。

在理解了 `for` 循环的三个要素之后，再来理解 `for` 循环的执行流程就比较清楚了。程序进入 `for` 循环语句之后，首先会执行初始化语句，完成必要的初始化工作。然后计算条件表达式的值，如果条件表达式的值为 `true`，则执行循环体语句，再执行更改语句，修改循环控制变量。接着又开始计算条件表达式的值，根据其值决定是否需要进行下一次循环：如果条件表达式的值为 `true`，则继续下一次循环；反之，则结束整个循环。`for` 循环控制结构的执行流程如图 4-5 所示。

为了深入了解 `for` 循环的机制，我们来看一个实际的例子。还记得小时候上数学课，老师让我们计算从 0 到 100 的所有整数的和吗？我总是从 0、1、2 开始逐个计算，所以总是落在那些懂得巧妙计算方法的同学后面。现在学习了 `for` 循环，使用 `for` 循环挨个计算也同样很快，代码如下：

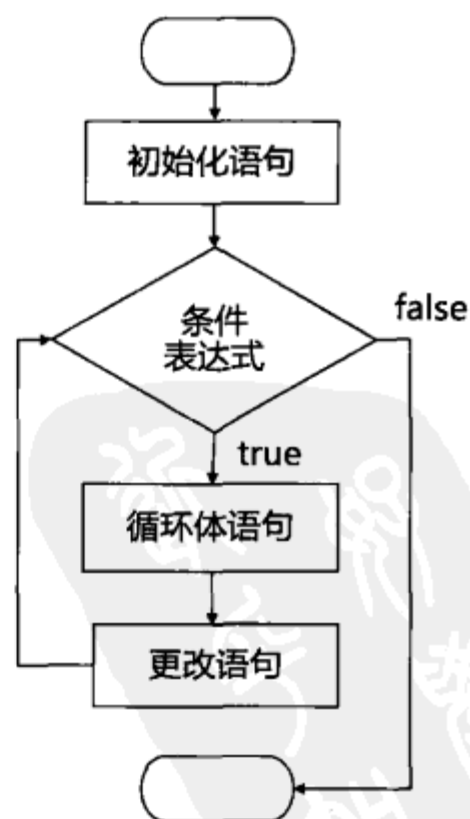


图 4-5 `for` 循环的执行流程

```

// 计算从 1 到 100 之间所有整数的和
// 记录和值
int nTotal = 0;
// 使用 for 循环，逐个将 1 到 100 之间的整数跟 nTotal 相加
for( int i = 1 ; i <= 100 ; ++i )
{
    // 将整数相加
    nTotal += i;
}
cout<<"1 到 100 之间所有整数的和是"<<nTotal<<endl;

```

在这段程序中，使用 for 循环快速遍历了 1 到 100 之间的所有整数并完成了求和任务。在这个 for 循环中，首先在初始化语句中定义一个循环控制变量 *i* 并赋初始值为 1，也就是循环的起点。然后，进行条件表达式的判断，这时 *i* 的值为 1，它是小于 100 的，所以条件表达式的值为 true，开始执行循环体。在循环体中，将 *i* 的值加到 nTotal 变量，nTotal 就记录了所有累加的结果。接着，for 循环开始执行更改语句，对 *i* 进行自增运算，将 *i* 的值变为 2。完成更改语句之后，for 循环又开始用新的 *i* 值进行条件表达式的判断，这时条件表达式还是 true，可以继续执行。如此反复，直到 *i* 的值变为 101 时，条件表达式的值为 false，整个 for 循环执行才结束。

通过这个例子可以发现，在 for 循环的初始化语句中，可以设定循环的起始值；在条件表达式中，可以设定循环的终止值，这样就给 for 循环设定了一个循环的范围：“从……到……”。所以这种拥有某个特定循环范围的循环场景最适合用 for 循环来表达。

4.3.4 循环控制：break 和 continue

循环周而复始，但也有被打破的时候。如果遇到阴天，太阳公公要休假，我们就没法看到太阳升起；遇到堵车，公共汽车也不是那么准时。为了描述循环中的例外情况，C++ 提供了两个关键字：break 和 continue。使用这两个关键字，可以轻松应对循环的例外情况，为循环提供应急预案。

1. break

在介绍 switch 条件选择结构的时候已经介绍过 break 关键字，当它用在循环结构中时，表示跳出当前循环，继续执行循环控制语句之后的下一条语句。下面用 break 关键字改写上文中的收支统计程序。

```

int nTotal = 0;
int nInput = 0;
do
{
    cout << "请输入你的收入或支出：";
    cin>>nInput;
}

```

```

        if( 0 == nInput )
            break;
        nTotal += nInput;
    } while ( true );

```

这里，我们使用 do...while 循环构造了一个条件表达式始终是 true 的“死循环”。啊，死循环！？不要大惊小怪，有 break 关键字的帮忙，这个“死循环”还可以活过来，照样正常工作。这样改写后，整个循环语句可以无限地接受用户的输入，当然，只要你有那么多收入。但是当用户想结束输入的时候，可以输入 0 满足“0 == nInput”这个条件，执行 break 关键字，跳出当前循环，结束整个循环的执行，这样一个“死循环”又变活了。当在某种条件下需要提前结束整个循环的执行时，可以采用 break 关键字跳出当前循环，继续执行循环控制结构后面的语句。

2. continue

break 关键字是在某种条件下跳出循环，而 continue 关键字则是在某种条件下结束本次循环体，然后判断条件是否满足进行下一次循环。还是使用上面的收支统计例子，如果一个大款，对小钱根本不在乎，只想统计大于 1 000 的收入和支出，则可以用 continue 关键字将程序改写如下：

```

// 大款的收支统计程序
int nTotal = 0;
int nInput = 0;
do
{
    cout << "请输入你的收入或支出：";
    cin>>nInput;
    if( 1000 < nInput )
        continue;
    nTotal += nInput;
} while ( 0 != nInput );

```

在这个大款的收支统计程序中，nInput 接收用户输入后判断其值是否小于 1 000，如果小于 1 000，则执行 continue 关键字，跳过后面的求和语句“nTotal += nInput;”，而直接跳转到对条件表达式“0 != nInput”的计算，判断是否可以开始下一次循环。值得注意的是，在 for 循环中，执行 continue 后，并没有跳过控制条件变化的更改语句，仍然将被执行，然后再计算条件表达式，尝试下一次循环。

虽然 break 和 continue 都是在某种条件下跳出循环，但是两者有本质的差别：break 是跳出整个循环，立刻结束循环语句的执行；而 continue 只跳出本次循环，继续执行下一次循环。图 4-6 展示了 break 和 continue 之间的区别。

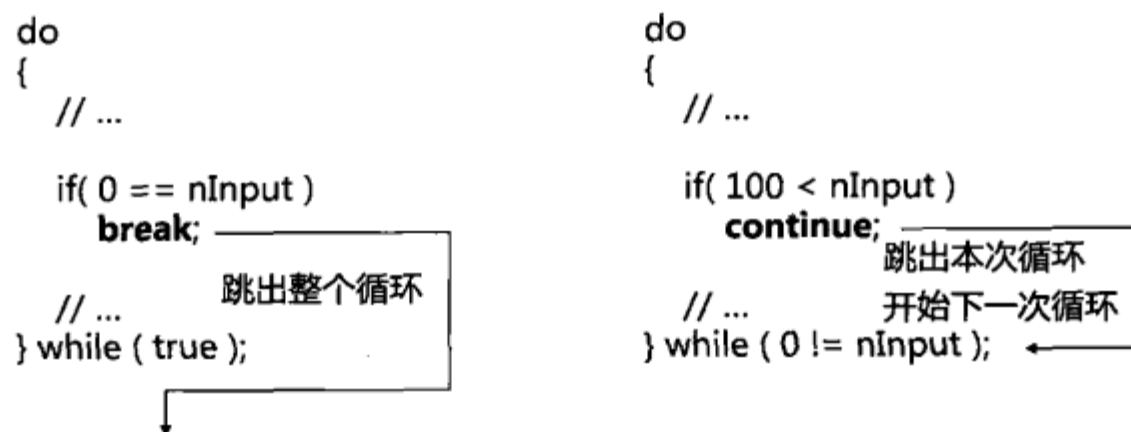


图 4-6 break 和 continue 之间的区别

4.4 从语句到程序

4.4.1 程序是控制语句串联起来的语句

了解了表达式和各种语句之后，就相当于掌握了写作文要用到的词语和句子，但是，光有词语和句子是无法构成一篇通顺的有意义的文章的。要完成一篇文章，还需要知道文章的结构，再按照一定的顺序和结构安排词语和句子，这样才能写出一篇有意义的文章，否则只会是一堆杂乱无章的句子。编写程序就像写文章一样，在掌握了基本的语句之后，也要知道程序有哪些结构，如何将这些零散的语句串联起来描述一个完整的计算过程，从而将语句编织成程序。

下面就以简单的计算两个数之和的程序为例，来看看程序中的语句是如何串联起来的。

```
// 计算两个数的和
int _tmain(int argc, _TCHAR* argv[])
{
    cout<<"请输入两个整数: "<<endl;

    int a,b;
    cin>>a>>b;
    int c = a + b;
    cout<<a<<"和"<<b<<"的和是: "<<c<<endl;

    return 0;
}
```

我们知道，所有程序都是从入口函数_tmain()开始执行的。进入入口函数后，程序便开始从上往下依次执行程序中的每一条语句，程序的执行流程在整个执行过程中不会发生改变。我们称这样的执行顺序为程序的顺序结构。到现在为止，所有的例程都是顺序执行的。顺序结构的程序可以完成一些简单的只需一步步顺序执行的任务。

最佳实践：通过单步调试来了解程序的执行过程

当程序结构比较复杂时，程序的执行过程分析起来就比较困难，这时可以在程序最开始的地方设置一个断点，然后用单步调试的方法，清楚地查看整个程序的执行过程。

现实世界是复杂的，很多问题并不是只用顺序步骤就可以解决。还是以上面的求两个数和的程序为例子，如果要求输入的数必须大于 100，则只有大于 100 的两个数才能计算它们的和。这时就需要判断输入的数是否大于 100，如果满足条件，就计算和；如果不满足，就提示用户输入的数不满足条件，无法计算。在这个过程中，需要根据条件决定程序执行的路径，这就是选择结构。选择结构可根据不同条件做出决策，选择不同的执行路径，实现不同的功能。例如：

```
int a,b;
cin>>a>>b;
// 选择结构，根据条件不同，执行的路径也不同
if( a < 100 || b < 100 )
{
    // 如果用户输入的数据不满足条件，
    // 则提示用户输入错误
    cout<<"请输入两个大于 100 的数."<<endl;
}
else
{
    // 如果用户输入的数据满足条件，
    // 则计算结果并输出
    int c = a + b;
    cout<<a<<"和"<<b<<"的和是："<<c<<endl;
}
```

再来看另外一种情形：要求输入 100 个数后计算它们的和。如果采用顺序结构，那么就要用到 100 个输入语句来接收用户的输入数据，这样就会显得很烦琐。实践证明，接收用户输入的动作是可以重复进行的，完成一次输入后可以再次输入，这样形成一个循环往复的过程，最终通过简单的语句来完成复杂的输入。像这种可重复多次循环执行的语句，就形成循环结构。循环结构主要用于一些需要反复执行的语句，这些语句完成的功能需要多次重复实现。例如：

```
// 循环结构
int nArray[100];

// 用循环结构多次接收用户输入
for( int i = 0; i < 100; ++i )
{
    cin>>nArray[i];
}
```

```
// 用循环结构，计算多个数的和
int nTotal = 0;
for( int i = 0; i < 100; ++i )
{
    nTotal += nArray[i];
}

cout<<"输入的 100 个数的和是："<<nTotal<<endl;
```

顺序结构、选择结构、循环结构是三种基本的程序控制方式。通过这三种方式的组合，可以描述复杂的程序执行流程。如果说程序中的语句是一颗颗璀璨夺目的珍珠，那么这三种控制结构就是将珍珠串联起来的丝线，只有通过丝线的串联，零散的珍珠才能构成一串美丽的项链。正是依靠这三种控制结构将分散的语句串联起来，表达一定的运算逻辑，才形成了一个完整的程序。

4.4.2 豪华工资统计程序

还记得本章开始那个让我们束手无策的工资管理系统吗？现在，我们已经有能力来完成这个工资统计系统了。

```
// 豪华工资统计程序
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个常量，表示总员工数
    const int MAX_NUM = 1000;
    // 保存所有工资数据的数组
    int nSalary[MAX_NUM];

    // 输入每位员工的工资
    int nTemp = 0;           // 临时变量，暂存用户输入的数据
    int nIndex = 0;         // 输入的序号
    // 使用循环控制语句接收用户的多次输入
do
{
    cout<<"请输入员工"<<nIndex<<"的工资："<<endl;
    cin>>nTemp;
    // 只要员工不是卖身为奴，他就不可能欠你钱
    // 所以输入的工资不能为负值，使用条件控制语句
    // 对输入错误进行处理，提示重新输入
    if ( nTemp < 0 )
    {
        cout<<"你开玩笑？这位员工欠你钱？请重新输入！"<<endl;
        continue;
    }

    // 将合法的数据保存到数组中，开始下一次输入
    nSalary[nIndex] = nTemp;
    ++nIndex;
```

```

    } while ( nIndex < MAX_NUM );

    // 计算平均工资
    float fAverageSalary;
    int nTotal = 0; // 工资总和
    // 使用循环控制结构对遍历数组中的数据进行统计
    for( int i = 0; i < MAX_NUM; ++i )
    {
        nTotal += nSalary[i];
    }
    // 平均工资 = 工资总和 / 员工数
    fAverageSalary = (float)nTotal / MAX_NUM;
    cout<<MAX_NUM<<"位员工的平均工资是"<<fAverageSalary<<endl;
    // 计算最高工资和最低工资...

    return 0;
}

```

以上这个豪华的工资统计程序综合应用了多种控制结构：首先定义保存工资的数组，然后接收用户输入的数据并保存到数组中，最后利用数组中的数据统计平均工资。在接收用户数据输入的时候，使用条件控制结构来对用户输入的合法性进行判断。如果用户输入合法，就将其保存到数组中；如果输入不合法，就提示用户重新输入。统计平均工资时，使用循环控制结构遍历数组中保存的每一个数，再将它们相加到一起成为工资总和后除以员工数就得到了平均工资。

三种控制结构和各种语句的综合运用，完美地解决了这个问题。现在，就是我们美梦成真的时刻！

用函数封装程序功能

在完成豪华的工资统计程序之后，我们信心倍增，开始向 C++ 世界的更深远处探索。

现在，可以使用各种数据类型和程序流程控制结构来编写完整的程序了。但是，随着要处理的问题越来越复杂，程序的代码也越来越复杂，主函数也越来越长了。这就像将所有东西都堆放到一个仓库中，随着东西越来越多，仓库慢慢就被各种东西堆满了，显得杂乱无章，管理起来非常困难。面对一个杂乱无章的仓库，聪明的仓库管理员提供了一个很好的管理办法：将东西分门别类地装进箱子，然后有序地堆放各个箱子。

这个好方法也可以用到程序设计中，把程序装进箱子，让整个程序结构清晰。

5.1 函数就是一个大“箱子”

当要处理的问题越来越复杂，程序越来越庞大的时候，如果把这些程序代码都放到主函数中，将使得整个主函数异常臃肿，这样会给程序的维护带来麻烦。同时，要让一个主函数来完成所有的事情，几乎是一项不可能完成的任务。在这种情况下，可以根据“分而治之”的原则，按照功能的不同将大的程序进行模块划分，具有相同功能的划分到同一个模块中，然后分别处理各个模块。函数，则成为模块划分的基本单位，是对一个小型问题处理过程的一种抽象。这就像管理一个仓库，总是将同类的东西放到同一个箱子中，然后通过管理这些箱子来管理整个仓库。在具体的开发实践中，我们先将相对独立的、经常使用的功能抽象为函数，然后通过这些函数的组合来完成一个比较大的功能。举一个简单的例子：看书看得肚子饿了，我们要泡方便面吃。这其实是一个很复杂的过程，因为这一过程中我们先要洗锅，然后烧水，水烧开后再泡面，吃完面后还要洗碗。如果把整个过程描述在主函数中，那么主函数会非常复杂，结构混乱。这时就可以使用函数来封装整个过程中的一些小步骤，让整个主函数简化为对这些函数的调用，如图 5-1 所示。

使用函数封装功能的另外一个重要优势是函数可以被不同的模块调用，从而实现代码的复用。这就像一个箱子既可以放在这个仓库，也可以放在另外一个仓库。例如，泡面可以调用烧水这个函数，同样煮饭也可以调用烧水这个函数，这样只需要一个烧水函数，既省时又省力。

现在，主函数就成了一个大仓库，而其中的各个实现具体功能的函数就是仓库中的一个箱子。作为一个聪明的仓库管理员，就是先将东西按照具体功能进行切分，然后分放到各个箱子中，最后将这些箱子有序地堆放到仓库中。从而完成一项比较复杂的任务。

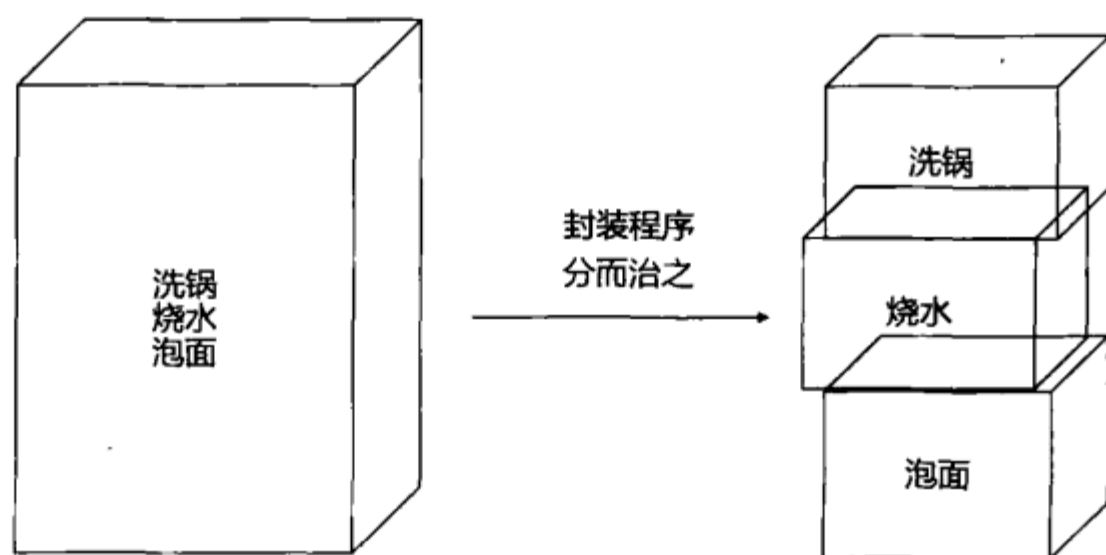


图 5-1 将程序封装到箱子，分而治之

5.1.1 函数的声明和定义

提问：把大象装到冰箱中需要几步？

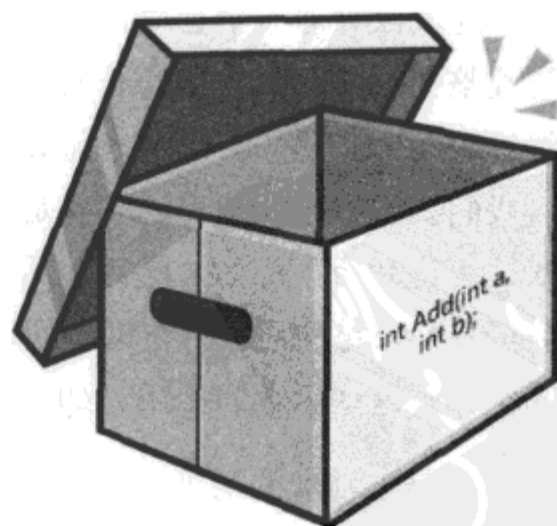
回答：需要三步。第一，打开冰箱；第二，把大象放进冰箱；第三，关上冰箱。

提问：把一个程序放进箱子中需要几步？

回答：需要两步。第一，声明一个函数；第二，定义这个函数。

没错，把一个函数放进箱子中比把大象放进冰箱还要简单。当分析一段长的程序代码时，往往会发现一些代码所实现的功能相对比较独立。将程序中这些相对比较独立的功能代码组织到一起，用函数对其进行封装，也就是将一个较长的程序分放到各个函数中。

要装东西，先得准备好箱子。为了找到具体的箱子，需要给箱子贴上标签。这个标签就是 C++ 中的函数的声明，如图 5-2 所示。



在 C++ 中，声明一个函数的语法格式如下：

返回类型标识符 函数名 (形式参数表);

图 5-2 声明一个函数就好比为为箱子贴上标签

例如，下面的代码定义了一个加法函数，用来计算两个整数的和。

```
int Add(int a, int b);
```

对照声明函数的语法格式，下面来看看这个加法函数声明中的各个部分。

1. 返回类型标志符

函数在执行完任务后，往往需要给它的调用者一个返回值，表示函数运行的结果或者其他意义。函数的返回类型标识符就是函数返回值的类型。在这个加法函数中，返回值类型为 `int`，表示该函数在完成加法计算后，将计算的结果作为返回值返回给它的调用者。如果函数只执行一些动作，无须返回值，则可以使用“`void`”作为返回值的类型。

2. 函数名

函数名就是为了标志一个函数而取的名字，就像给箱子贴上的标签一样，通过该函数名找到这个函数。函数的命名规则跟变量的命名规则相同。如果说变量命名重在说明它“是什么”，那么函数的命名则重在说明它要“做什么”。例如，在上面的例子中，函数实现的是两个数的加法运算，我们就将这个函数命名为 `Add`。

3. 形式参数表

在调用函数的时候，往往要进行函数间的数据交换，向函数传入或者从函数传出一些数据。函数的参数就是用来进行数据交换的，而形式参数表是对函数参数的描述。形式参数表的语法格式如下：

数据类型 1 参数名 1, 数据类型 2 参数名 2...

在上面的加法函数中，“`int a, int b`”就是形式参数表。通常情况下，函数调用者使用形式参数表把数据传递给函数，但在特殊情况下(使用指针或引用)也可以用来从函数向函数调用者传递计算结果等。在使用函数的形式参数表时，有以下几点需要注意的地方。

(1) 形式参数要有明确的数据类型。

函数参数的定义与定义变量类似，总是先写参数的数据类型，然后写参数的名字。如上面例子中的“`int a`”，其中 `int` 是参数的数据类型，`a` 是参数的名字。要向函数传递多个数据时，可以在形式参数表中定义多个参数，各个参数之间用逗号间隔。例如，上面例子中的“`int a, int b`”就定义了两个参数 `a` 和 `b`。在形式参数表中，每个参数必须有明确的数据类型说明，两个相同数据类型的参数不能使用同一个数据类型说明符。例如，在上面的例子中，虽然 `a` 和 `b` 的数据类型相同，但是形式参数表不能写成“`int a, b`”。

(2) 形式参数可以有默认值。

在定义变量时可以给定变量的初始值，同样，在定义参数时也可以给参数一个初始值，这个参数就称为默认参数。例如，可以写一个函数来判断某个分数是否及格。为了跟及格分

数线进行比较，这个函数可以有一个参数用来表示及格分数线。在大多数情况下，及格分数线为 60，这时就可以用 60 作为参数的默认值：

```
bool IsPassed( int nScore = 60 );
```

使用默认参数，可以给函数调用带来很大的灵活性。大多数情况下，可以在调用函数的时候省略参数，直接使用该参数的默认值对函数进行调用。但是在一些必要的情况下，又可以使用其他的数值改变默认参数，让其他参数值来调用这个函数。当声明一个带有默认参数的函数时，这些默认参数必须位于形式参数表的末尾，不能在形式参数表的开始或者中间定义一个默认参数，例如：

```
bool max( int a = 0, int b ); // 错误的形式，默认参数不能在形式参数表的开始位置
bool max( int a, int b = 0 ); // 正确的形式，默认参数在形式参数表的末尾位置
```

(3) 没有形式参数时可以用 void 代替。

一个函数的形式参数并不是必须的，有很多函数没有形式参数表，这时既可以将形式参数表留空，也可以用 void 代替形式参数表，表明这个函数没有形式参数表。例如：

```
// 将形式参数表留空
void DoSomething()
// 或者使用 void 代替形式参数表
void DoSomething(void);
```

在这个函数声明中，将形式参数表留空，或者使用 void 作为函数的形式参数，表明 DoSomething() 函数没有形式参数表，也就是说，无须跟函数的调用者通过函数参数的形式进行数据交换，只是单纯地完成某项功能而已。

完成函数的声明，只是将程序装进箱子的第一步，第二步就是要完成函数的定义，将具体的程序代码放到箱子中。函数的定义往往是紧接着函数的声明进行的，其语法格式如下：

```
返回类型标识符 函数名(形式参数表)
{
    // 函数定义
}
```

函数的定义紧接着函数的声明进行，用一对花括号“{}”括起来的部分就是一个函数的定义，也称为函数体。我们在函数的定义中完成函数的具体功能。例如，可以这样来定义上面声明的 Add() 函数，对两个数进行相加并返回它们的计算结果。

```
// 计算两个数的和
int Add( int a, int b )    // 函数声明
{
    // 函数定义
    int nResult = a + b;
```

```
// 函数返回值  
return nResult;  
}
```

在这段代码中，函数声明之后用花括号括起来一段代码就是 Add()函数的函数体。如果 Add()函数有返回值，还必须在函数体最后使用 return 关键字将结果返回给函数调用者。

“return”即“返回”的意思，当函数体内的代码执行到 return 语句时，函数即告结束，并将结果返回给函数的调用者。return 返回的结果类型必须和函数声明中的返回值类型一致。如果 return 后面还有代码，则后面的代码不会被执行。根据程序的执行情况，同一个函数可以有多个 return 语句，用于不同的情况下返回不同的结果。当然，对于不需要返回结果的函数，可以不写 return 语句，函数体在执行完所有代码后自然结束。

上面例子中的 Add()函数的函数体只有两句，第一句是计算参数 a 和 b 的和并保存到临时变量 nResult 中，第二句是将 nResult 作为计算结果，使用 return 语句返回给函数调用者。这样，该函数体就实现了计算两个数之和的功能。

最佳实践：声明和定义相分离

在实际的开发活动中，有些程序是非常复杂的，它们动辄有上千万行代码，往往由多个源代码文件构成。根据声明和定义相互分离（接口和实现分离）的原则，源代码文件通常被分成两类：一类文件主要用来定义函数及声明类，这类文件称头文件，通常以.h 为后缀；另外一类文件则用来定义函数和类，实现函数和类的具体功能，这类文件称为源文件，多以.cpp 为后缀。

因为函数声明通常在头文件中，所以可以通过引入某个头文件来引入其中声明的函数。同时，也提示我们，在开发实践中需要将函数的声明和定义放在不同的文件中，以实现接口和实现的相互分离。

5.1.2 函数调用机制

在学习编写函数之前，首先要了解函数的调用机制，学会如何调用一个已经存在的函数。世界上已经有很多函数，我们可以直接调用这些函数来完成日常任务。在实际的开发中，可供调用的现有函数主要有编译器提供的库函数、Windows API 及第三方提供的函数库等。通过调用他人的函数，可以复用他人的开发成果，在其开发成果的基础上，实现快速开发，如图 5-3 所示。

有了别人提供的函数，就可以调用这些函数来完成自己的功能。两个函数之间的关系是调用与被调用的关系，把调用其他函数的函数称为主调函数，被其他函数调用的函数称为被调函数。一个函数是主调函数还是被调函数并不是绝对的，要根据其所处的相对位置而定：如果一个函数内部有函数，则相对其内部的函数它就是主调函数；如果它的外部有函数，则相对其外部函数它就被调函数。

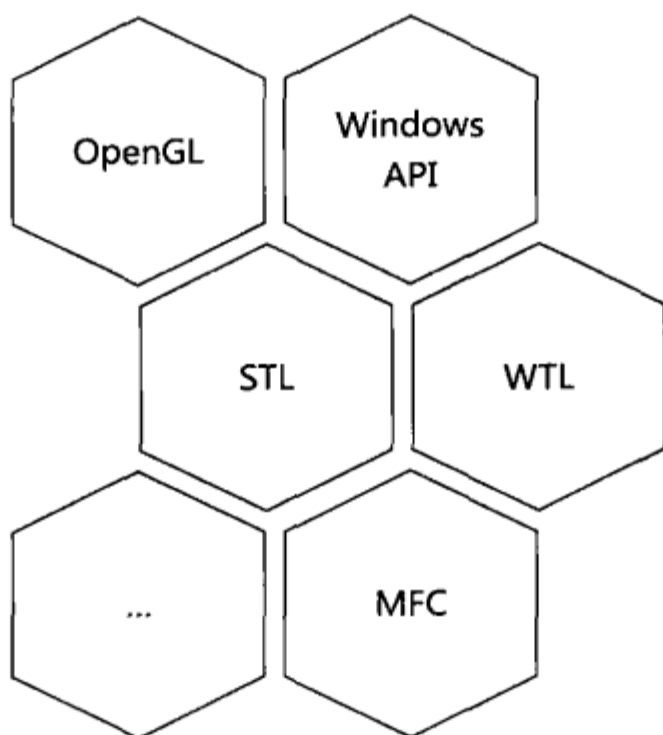


图 5-3 天上掉下个函数库

可以通过下面的形式来调用函数库或者自己定义的函数：

返回值变量 = 函数名(参数);

这就是一条简单的函数调用语句，其中，用返回值变量来保存函数的返回值，如果函数没有返回值，则这个变量可以省略。更重要的是，可以用“函数名()”的形式来实现对一个函数的调用，如果这个函数有参数，就将参数分别放到函数名之后的括号中，形成函数调用的实际参数。例如：

```
int nResult = Add(1, 2);
```

这行代码实现了对 Add() 函数的调用，这里，我们向 Add() 函数传递两个参数 1 和 2，然后将 Add() 函数执行完成后的返回值保存到 nResult 变量中。这样，就由 Add() 函数来完成加法运算的具体功能，我们只需要简单地调用一个函数，就得到了加法运算的结果。至于具体的运算过程，可以交给 Add() 函数去处理，无须我们操心。函数调用的本质就是把一部分工作交给其他人去完成。

一个简单的函数调用就可以完成很多事情，那么，一个函数调用到底是如何进行的呢？在程序执行过程中，如果遇到了对其他函数的调用，则需要执行一系列的动作。首先，暂停主调函数的执行，保存现场，传递参数给被调函数；其次，从被调函数的入口地址开始进入被调函数，将控制权交给被调函数，并开始执行被调函数代码；当被调函数执行结束返回时，恢复先前保存的现场，控制权交还给主调函数并继续主调函数的执行。下面再来看一个实际的例子。

```
// 定义一个加法函数  
int Add(int a, int b)
```

```

{
    int nResult = a + b;

    return nResult;
}

// 在主函数中调用加法函数
// 这时的主函数可以称为主调函数
int _tmain(int argc, _TCHAR* argv[])
{
    int a = 1;
    int b = 2;
    // 调用 Add() 加法函数
    // 这时的 Add() 加法函数可以称为被调函数
    int nResult = Add(a, b);
    cout<<a<<" + "<<b<<" = "<<nResult<<endl;

    return 0;
}

```

这是表示_tmain()函数对实现加法运算的 Add()函数的调用。当运行程序时，从_tmain()函数开始，首先定义 a、b 两个变量并对其赋值，然后以 a 和 b 为参数调用 Add()函数来计算这两个数的和。虽然从表面上我们并不能看到函数调用的实现细节，但是在背后函数调用却做了很多事情。首先用 a 和 b 对 Add()函数的两个形式参数赋值，将控制权交给 Add()函数，开始执行 Add()函数。在 Add()函数中，两个形式参数 a 和 b 经过赋值后，其值分别为 1 和 2，这就完成了从主调函数传递数据给被调函数的过程。然后 Add()函数开始执行具体的运算过程，也就是计算两个形式参数的和。最后还需要用 return 关键字将计算结果返回，作为整个函数调用表达式的结果，主调函数可以利用这个结果进行进一步的赋值或者运算。被调函数返回后，控制权重新交还给主函数。主函数继续执行，将函数的返回值赋值给 nResult 变量，并将计算结果输出。整个调用过程如图 5-4 所示。

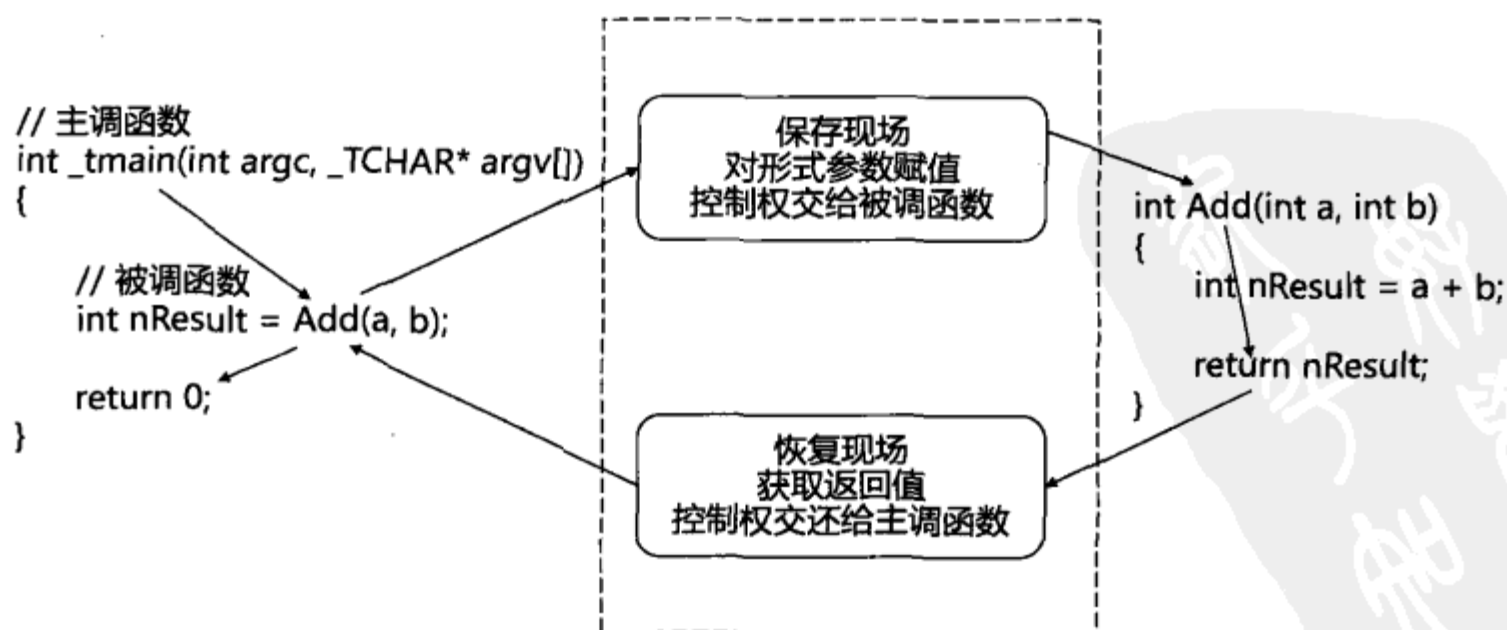


图 5-4 函数调用的执行流程

在图 5-4 中，箭头的方向代表整个 Add()函数被调用的流程，虚线框包围的部分是系统为了实现函数调用而额外做的幕后工作。

我们知道函数的参数可以有默认值。那么，对于有参数默认值的函数又该如何调用呢？带默认参数的函数的调用跟普通函数的调用一样，只是它可以使用参数的默认值，使得带默认参数的函数更加灵活。在调用的时候可以省略具有默认值的参数，直接使用参数的默认值对函数进行调用，例如：

```
bool bPassed = IsPassed(); // 直接使用参数的默认值，相当于调用 IsPassed( 60 )
//我们也可以把它当做一个普通函数，给定具体的参数值进行调用，例如：
bool bPassed = IsPassed( 83 ); // 使用具体的参数值代替参数的默认值
```

当然，并不是仅使用调用和被调用这两层关系就能把问题描述清楚。很多情况下，被调用者往往同时也是调用者，它也会调用其他的函数来完成更加复杂的功能。反映到 C++语言中，就是 C++不仅支持主调函数调用被调函数，还支持在被调函数中继续调用其他函数，这就是函数的嵌套调用。例如，“泡面”函数需要调用烧水函数，而“烧水”函数又需要调用“打开电源”函数，等等，这样便形成了函数的嵌套调用。嵌套函数调用的意义就是不断细化与分解一个较大任务。

为了理解函数的嵌套调用，下面来看一个计算平方和的程序。

```
// 计算平方函数
int Power( int n )
{
    return n*n;
}
// 计算平方和函数
int PowerSum( int a, int b )
{
    return Power(a) + Power(b);
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 调用求平方和函数
    int nResult = PowerSum(2,3);

    return 0;
}
```

我们知道，求平方和的方法是先求两个数的平方，然后再进行相加运算以求得两个数的平方和。按照这样的思路，首先将求平方和的任务分解为求平方及求和两个子任务，也就是 Power()和 PowerSum()两个子函数。在程序中执行具体的计算过程时，在主函数中调用 PowerSum()函数，而 PowerSum()函数又嵌套调用 Power()函数，这样就将一个比较复杂的问题通过不断细化和分解，最后转化为比较简单的问题，如图 5-5 所示。这正好也反映了程序

设计中的“自顶向下，逐步求精”的设计思想。

知道更多：自顶向下，逐步求精

在现实世界中，有许多任务是很复杂的，当无法在一个函数中解决所有问题时，可以考虑采取“分而治之”的原则，将较大的任务分成多个较小的任务，而这些较小的任务可以在一个函数中完成，这就是函数的分解封装。

较小的任务又可以根据需要进一步细分，这就是函数的嵌套。就像盖一座大楼，首先要将大楼分成很多层，然后每层又分成很多个房间，而每个房间又由很多砖头构成。这种将大问题逐渐分解的程序设计方法，被称为“自顶向下，逐步求精”的设计方法。也就是说，在写一个程序时，先应该考虑整体的结构，然后再不断细化，最终完成整个任务。

“自顶向下，逐步求精”是结构化程序设计的精髓，这种方法符合人类解决复杂问题的普遍规律，可以显著提高软件开发的效率。同时，用先全局后局部、先整体后细节、先抽象后具体的“逐步求精”的过程开发出来的程序有清晰的层次结构，更容易阅读和理解。

随着任务复杂度的增大，当程序变得非常大时，想一次性理清整个程序的脉络是不可能的。这时一种新的程序设计方法产生了，这就是面向对象的设计方法。将会在稍后的章节中做更加详细的介绍。

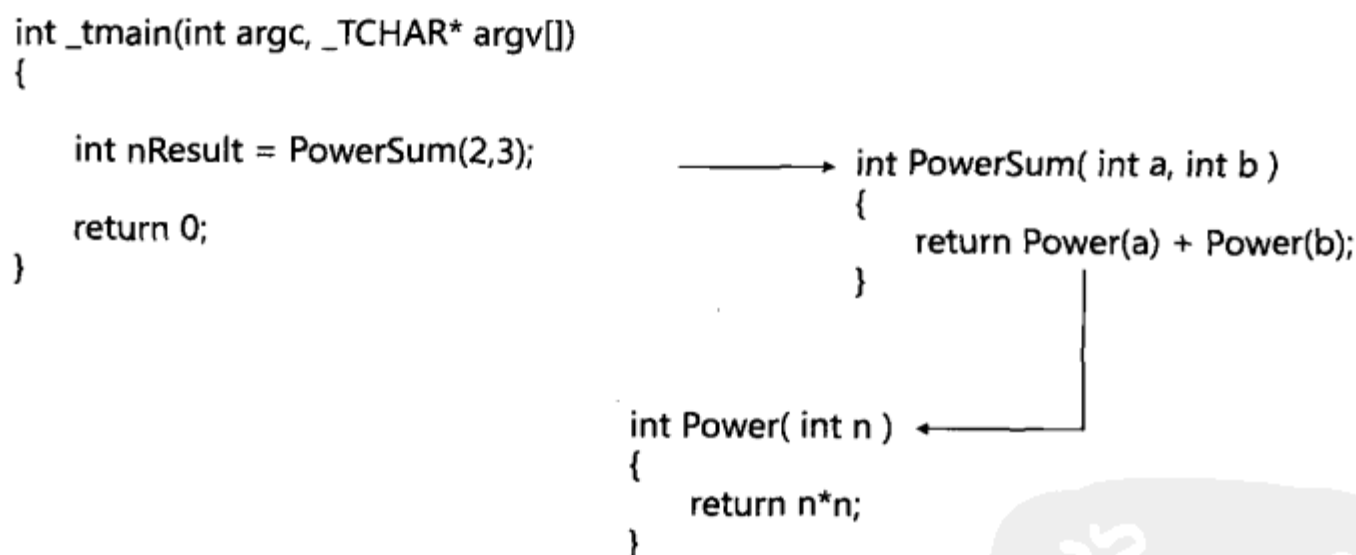


图 5-5 平方和程序的函数嵌套调用

5.1.3 函数的声明与函数调用

在学会了函数调用后，是不是就可以任意调用任何函数了呢？

当然不是。就像我们在使用箱子前，必须要知道箱子存放在哪个仓库中，箱子上的标签是什么，这样才能找到箱子，从而使用箱子封装的功能。在函数调用中也是这样，当主调函数调用一个函数时，必须能够找到被调函数，这就要求函数在被调用之前都需要进行声明。

例如：

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 调用函数库中的 asin 函数
    double pi = asin(1.0) * 2;
    cout<<"pi = "<<pi<<endl;

    return 0;
}
```

在 Visual C++ 中编译这段代码时，编译器会提示如下编译信息：

```
error C3861: "asin": 找不到标识符
```

这是因为主函数在调用 `asin()` 函数之前，并没有声明 `asin()` 函数，所以编译器把它当成了无法找到的标识符。那么如何解决这个问题呢？很简单，因为 `asin()` 函数声明在 `math.h` 头文件中，所以只要在程序开始引入 `math.h` 头文件，让主函数在调用 `asin()` 函数时找到 `asin()` 函数就可以解决问题了。

```
#include "stdafx.h"
#include <iostream>
// 引入 math.h 头文件
// 可以使用其中定义的各种数学函数
#include <math.h>
// ...
```

因为经常使用头文件来引入其他函数的声明，所以可以记住这样一条规则：想要使用某个函数，就先引用它所在的头文件。

除了使用头文件来引入库函数的声明外，对于自己定义的函数，必须在这个函数被调用之前进行声明，否则，这个函数也会被当成找不到的标识符，产生编译错误。例如：

```
// 声明并定义 Add 函数
int Add( int a, int b )
{
    return a + b;
}

// 声明 Power 函数
int Power( int n );

// 声明并定义 PowerSum 函数
int PowerSum(int a, int b )
{
    return Add( Power(a), Power(b) );
}
```



```
// 定义 Power 函数
int Power( int n )
{
    return n * n;
}
```

在这段代码中，在 PowerSum()函数中同时调用了 Add()函数和 Power()函数，为了让 PowerSum()函数在调用时能够找到这两个函数，必须在 PowerSum()函数之前声明这两个函数，也就相当于 PowerSum()函数能够找到所有在它之前声明的函数并且可以加以调用。至于被调用函数的定义，可以在 PowerSum()函数之前，如 Add()函数的定义；也可以在 PowerSum()函数之后，例如 Power()函数的定义。函数声明和函数定义的分离，给程序带来了很大的灵活性，在创建比较大型的程序时，可以把多个函数声明在某个头文件中，而将其定义在另外的源文件中。这样在使用函数时，只需要引入一个头文件就可以同时使用多个函数。我们通常使用的库函数就是按照这种方式来组织的。

5.1.4 函数参数的传递

我们可以把东西放进箱子中，同样，通过函数参数的传递，也可以将数据放到函数箱子中。我们知道，函数是用来完成某个相对独立功能的模块。函数在完成这个功能的时候，往往需要外部数据的支持，这时就需要在调用这个函数时向它传递所需要的数据。例如，当调用一个加法函数时，需要向它传递两个数作为加数和被加数。在定义一个函数的时候，如果这个函数需要跟外部进行数据交换，就会在函数定义中加入形式参数表，通过形式参数表可以将数据从函数外部传入函数内部。例如，可以这样定义一个加法函数：

```
// 声明并定义 Add 函数
// 用形式参数表来表示这个函数需要跟外界进行数据传递
int Add( int a, int b )
{
    return a + b;
}
```

从 Add()函数的声明中可以知道这个函数需要两个整型数作为参数，所以可以在调用的时候给它传递两个整型数作为参数：

```
// 以 1 和 2 作为参数调用 Add() 函数
// 也就是向这个函数传递 1 和 2 这两个数据
int nResult = Add(1, 2);
```

我们将在函数调用时给出的参数称为实际参数，例如 1 和 2 就是实际参数。在进行函数调用的时候，系统会使用调用函数时给出的实际参数分别对函数声明中的各个形式参数进行赋值。例如，这里 1 和 2 两个参数分别会赋值给 Add()函数的两个形式参数 a 和 b，也就是说，在 Add()函数内部，a 和 b 两个变量的值这时就是 1 和 2，这样就通过实际参数和形式参数实现了从函数外部向函数内部传递数据，如图 5-6 所示。

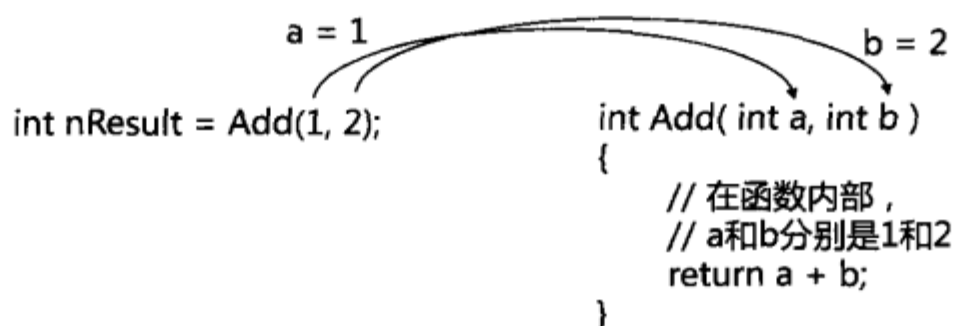


图 5-6 函数调用过程中的参数传递

在函数调用的时候，系统需要将实际参数复制给形式参数，如果要向函数内部传递一些比较大的数据，为了提高效率，则可以用传递指向这个数据的指针来代替传递整个数据。在函数内部，通过指针来访问数据，同样可以达到向函数内部传递大数据的目的。例如，可以通过传递数组的指针向函数传递一个数组。利用这种方式，可以用一个函数来封装“豪华工资统计”程序中的统计平均工资部分，并通过向它传递数组指针来实现整个工资数组的统计。

```
// 计算平均工资
float GetAverageSalary( int* pSalary, const int MAX_NUM )
{
    // 计算工资总和
    int nTotal = 0; // 工资总和
    for( int i = 0; i < MAX_NUM; ++i )
    {
        nTotal += pSalary[i];
    }
    // 平均工资 = 工资总和 / 员工数
    return (float)nTotal / MAX_NUM;
}
```

在 GetAverageSalary()函数的声明中，我们定义了两个形式参数，第一个表示需要统计的数组的指针，因为数组的指针并没有函数数组元素个数的信息，所以还需要用第二个形式参数来表示这个数组中元素的个数。这样，就可以在函数内部通过数组指针来访问整个数组，完成统计功能。

在调用 GetAverageSalary()函数的时候，按照函数的声明，可以以数组的首地址和数组元素个数作为实际参数对其进行调用。

```
// 定义数组
const int MAX_NUM = 100;
int nSalary[MAX_NUM];
// ...
// 调用 GetAverageSalary 函数
float fAverageSalary = GetAverageSalary(nSalary, MAX_NUM );
```

就这么简单，使用指针可以轻松地将数组这头大象放进函数这个小箱子。

5.1.5 函数的返回值

把大象放进箱子之后，最后还得把大象从箱子中取出来。数据也一样，通过参数传递可以把数据放到函数箱子中，那么又如何从箱子中取出数据呢？还记得在声明函数的时候，需要指明这个函数的返回值类型吗？只要一个函数的返回值类型不是 `void`，它就具有返回值，我们就是用函数的返回值来向函数的调用者提供函数的计算结果，从函数这个箱子中取出数据。下面还是以 `Add()` 函数为例：

```
int Add( int a, int b )
{
    return a + b;
}
```

在函数定义中，可以使用 `return` 关键字结束函数的执行并返回一个计算结果，这个结果就是从函数中取出的数据。换句话说，函数调用表达式的值就是从函数箱子中取出的数据，这个数据的类型就是函数的返回值类型。例如：

```
int nResult = Add(1, 2);
```

`Add(1, 2)` 函数调用表达式经过运算后，其返回值是一个整型数 3，3 就是从函数箱子中取出的计算结果数据。再利用这个数据对 `nResult` 变量进行赋值，这样从函数中取出的数据就被保留了下来。

使用函数调用表达式除了可以对变量赋值之外，还可以使用函数调用表达式对数值直接进行计算，将其应用在任何可以使用此类型数值的地方。例如：

```
// IsFinished() 函数调用表达式是布尔类型
// 直接与 true 进行逻辑运算
if( IsFinished() == true )
{
    // ...
}
// 函数调用表达式 Power(2) 和 Power(3) 是整型数值
// 直接用做另外一个函数的整型参数
int nResult = Add( Power(2), Power(3) );
```

从以上代码注意到，因为每个函数只有唯一的返回值，使用函数返回值只能从函数中取出一个数据，如果想要从函数中取出多个数据怎么办呢？

回想一下，我们是如何将一个大数据传入函数的？是的，使用了指针。利用指针的指代特性，可以在函数内部通过指针间接修改它所指向的数据，从而间接地实现数据的传出。不好理解吗？没关系，来看一个实际的例子。

```
// 输入员工的工资数据到工资数组中
int InputSalary(int* pSalary, const int MAX_NUM )
{
```

```

int nTemp = 0; // 临时变量，暂存用户输入的数据
int nIndex = 0; // 输入的序号
do
{
    cout<<"请输入员工"<<nIndex<<"的工资: "<<endl;
    cin>>nTemp;
    // 如果输入的是负数，表示输入工作结束，跳出循环
    if ( nTemp < 0 )
    {
        break;
    }

    // 将合法的数据保存到数组中，开始下一次输入
    // 通过修改传入的数组，实现数据的传出
    pSalary[nIndex] = nTemp;
    ++nIndex;
} while ( nIndex < MAX_NUM );

// 返回输入的数据总数
return nIndex;
}

```

InputSalary()函数用来输入员工的工资数据。我们利用指针向这个函数传递一个数组，并在函数内部将合法的输入保存到数组中，修改数组的数据。这样函数外部的数组就保存了函数内部的数据，间接实现了函数内部数据的传出。另外在这个函数中，还利用函数返回值从函数中得到了所有输入的数据总数。也就是说，利用函数返回值和函数参数从函数中传出数据这两种方式是可以同时使用的。当然，也可以使用 void 作为函数的返回值类型，从而使用函数的指针参数来完成数据的传入、传出，如图 5-7 所示。

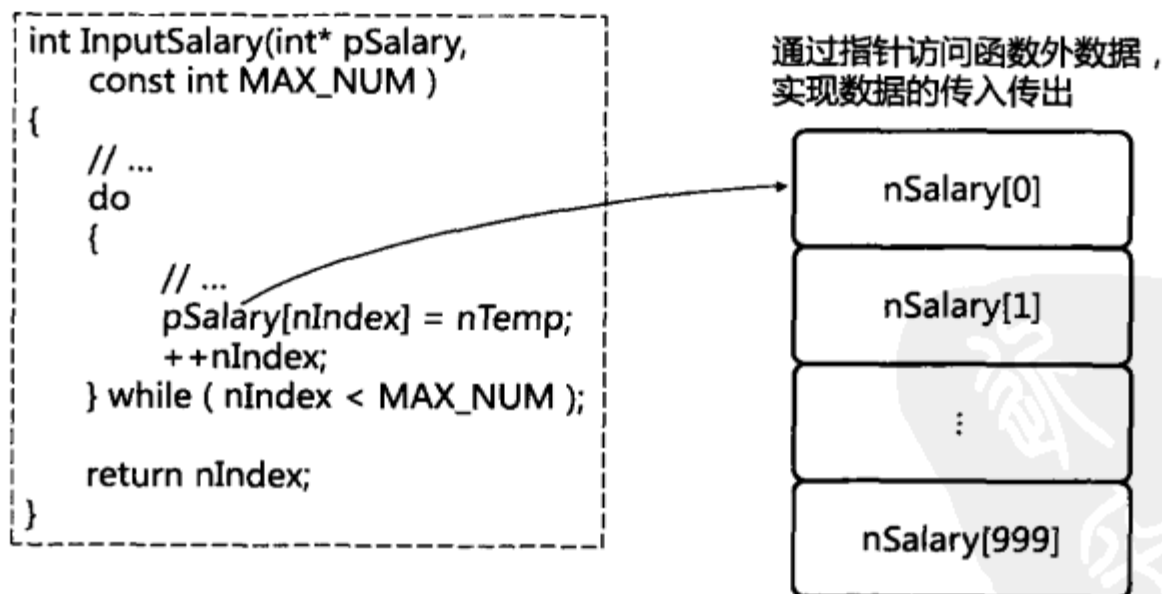


图 5-7 通过指针访问函数外部数据

现在，就可以利用 InputSalary()函数传出的数据进行下一步的计算了。

// 豪华工资统计程序

```

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个常量，表示数组的最大容量
    const int MAX_NUM = 1000;
    // 保存所有工资数据的数组
    int nSalary[MAX_NUM];

    // 输入每位员工的工资
    // 这里的 nSalary 作为 InputSalary() 函数的传出参数
    // 输入的数据保存在 nSalary 数组中，实现从函数内部传出数据到数组中
    // 函数返回输入的员工工资总数
    int nCount = InputSalary( nSalary, MAX_NUM );

    // 计算平均工资
    // 这时 nSalary 已经保存了输入的员工工资，
    // 这里作为 GetAverageSalary() 函数的传入参数，
    // 向其中传入保存后的员工工资数据
    float fAverageSalary = GetAverageSalary(nSalary, nCount );

    return 0;
}

```

经过“自顶向下，逐步求精”的功能分解，将主函数中相对独立的输入功能和统计功能分别封装到两个函数中，实现了将复杂程序分解装箱的工作。将程序装箱成函数后，之前比较复杂臃肿的主函数，现在只需要简单地调用这两个子函数就完成了所有功能。将程序装箱成函数，整个程序结构变得更加清晰，维护起来也更加容易。

5.2 内联函数

通过 5.1 节的学习我们知道，系统为了实现函数调用会做很多额外的工作：保存现场、对参数进行赋值、恢复现场，等等。如果函数在程序内被多次调用，且其本身比较短小，可以很快执行完毕，那么将导致系统花在函数调用上的时间远远大于函数执行的时间，造成极大的资源浪费，同时程序的性能也会降低。为了解决这个问题，C++提供了内联函数的机制，通过将函数代码内联到函数调用的地方，避免函数调用，从而提高这种短小的函数被重复多次使用的性能。

5.2.1 用体积换速度的内联函数

内联函数可以更形象地称为内嵌函数，它是 C++对函数的一种特殊修饰。当编译器编译程序时，如果发现某段代码在调用一个内联函数，它就不再去调用该函数，而是将该函数的代码整段插入当前函数调用的位置。这样就省去了函数调用的过程，提高了代码的执行效率。就像家里的电灯经常坏掉，经常要找修理工来修理一样，为了提高效率，干脆自己在家里养

一个修理工。这样只要电灯坏了，就可以立刻进行修理，省去了每次去找修理工的时间，提高了效率。但是这样也带来麻烦：家里不得不多养一个修理工。内联函数也有同样的烦恼，在提高效率的同时也带来了问题，因为内联函数需要在每个函数调用处插入该函数的代码，这将导致整个程序的体积增大。这就是典型的牺牲应用程序的体积换取应用程序的执行速度，如果你对程序的体积比较敏感，那么内联函数需要谨慎使用。

内联函数的声明比较简单，只要在普通函数声明前加上关键字 `inline` 就可以了，其语法格式如下：

```
inline 返回类型标识符 函数名(形式参数表)
{
    函数体语句;
}
```

例如，可以在 `Add()` 函数的声明前加上 `inline` 关键字对其进行修饰，将其变成一个内联函数，例如：

```
// 内联的 Add() 函数
inline int Add(int a, int b)
{
    return a + b;
}
```

内联函数的调用形式跟普通函数的调用形式没有任何区别，例如：

```
int _tmain(int argc, _TCHAR* argv[])
{
    Add(2, 3);

    return 0;
}
```

在 `Add()` 函数前加上 `inline` 关键字，`Add()` 函数就变为了内联函数，这样在 `_tmain()` 函数中调用 `Add()` 函数时，就不再是一次函数调用，编译器会直接把 `Add()` 函数的代码插入函数调用的位置，省略了中间的函数调用过程。

5.2.2 内联函数的使用规则

内联函数的使用往往是有利有弊的，只有遵循下面这些内联函数的使用规则，才能够尽量做到扬长避短，充分利用内联函数的优势来提高程序的性能。

1. 内联函数要短小精悍

由于使用内联函数会增加程序的体积，所以内联函数应该尽量做到短小精悍，一般不要超过 5 行。如果将一个比较复杂的函数内联，往往会导致程序的体积迅速膨胀，最后得不偿失。

2. 内联函数执行的时间要短

内联函数不仅要短小，还要执行时间短，这样才能体现内联函数的优势。如果函数执行的时间远大于函数调用的时间，那么通过内联函数所节省下来的函数调用时间也就无足轻重了。

3. inline 关键字仅仅是一种建议

inline 关键字仅仅是一种对编译器的建议，表明程序员对这个函数的处理意见。但是在某些特定的情况下，编译器将不理睬 inline 关键字，而强制让函数成为普通函数。这时编译器会给出相应的警告消息。所以，inline 关键字我们只是建议编译器将函数内联处理，至于到底是否进行内联处理还要看编译器的情况。

实际上，内联函数可以在一定程度上提高应用程序的性能，但是内联函数并不是解决性能问题的灵丹妙药，也并不是用得越多越好。很多时候，如果发现应用程序存在性能上的问题，应该更多地从应用程序的结构和设计上寻找问题的解决办法。内联函数，只是饭前的小甜点而已，偶尔尝一点还不错，但是并不能充饥。

5.3 重载函数

“嘿，编译器，我需要一个 Add() 函数，帮我搬一个 Add() 函数箱子过来。”

“老板，你到底要哪个 Add() 函数箱子？我这有好几个 Add() 函数呢！”

“好几个？我要的是可以做整型数的加法运算的 Add() 函数箱子啊！”

“我明白了，老板你要的是贴有 ‘int Add(int a, int b)’ 标签的函数箱子。”

“干吗搞这么复杂呢？”

“老板，这叫函数的重载，虽然箱子的名字相同，却是不同的箱子，分别用来装那些功能相近但是具体实现不同的函数。这样可以做到专门的箱子用做专门的用途。”

5.3.1 重载函数的声明

重载函数，就是让一个函数承载多种功能，具有多种含义。简单来讲，就是让同一个函数名表示多种意义。

大家可能会问，一个函数名表示一个意义不是很好吗，为什么要用一个函数名表示多个意义呢？这样不会造成混乱吗？

先别太早下结论，至于重载函数到底有没有必要，我们来看下面这个例子就明白了。在前面的章节中我们完成了一个 Add() 函数，表示两个数的求和。如果在调用 Add() 函数时给定

参数为两个整型数 2 和 3，则这个函数可以正常工作，返回正确的和值；如果给定的参数是两个浮点型数 2.5 和 3.7，很遗憾，这个函数就不能返回正确的和值了。

```
int nResult = Add( 2, 3 );           // 得到正确结果 nResult 为 5
int fResult = Add( 2.5, 3.7 );       // 得到错误结果 fResult 为 5.000000
```

在程序设计中，为了函数调用形式的统一，往往要求一个函数能够应对多种情况，处理多种类型的数据。例如，希望 Add() 函数既能计算整型数的和，也能计算浮点型数的和。在现实生活中，也同样存在相似的情形。我们常说“擦桌子”、“擦玻璃”、“擦皮鞋”，虽然同样都是“擦”这个动作，但是在不同的情况下，“擦”的对象各有不同。程序，是对现实世界的描述。为了描述这种根据上下文情况而有不同含义的动作，C++ 提供函数重载的机制来支持这种情形是非常必要的。有了函数重载机制，在开发的时候，就可以使用相同的函数名处理不同的数据，从而表示不同的含义。在编译的时候，编译器可以根据上下文（通常是参数个数和类型）来决定到底使用哪个具体的函数，以统一的形式实现对不同数据的处理，这就是函数重载的意义。

准确地讲，两个函数，因为实现的功能相似，所以取相同的函数名，但是参数的个数或类型不同，这就是函数重载，而这两个函数就被称为重载函数。

下面来看看如何使用函数重载来解决上文中 Add() 函数不支持浮点型数的问题。

```
// 计算整型数和的 Add() 函数
int Add( int a, int b )
{
    cout<<"int Add( int a, int b )被调用！"<<endl;
    return a + b;
}
// 计算浮点型数和的 Add() 函数
float Add( float a, float b )
{
    cout<<" float Add( float a, float b )被调用！"<<endl;
    return a + b;
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 因为参数是整型数
    // 调用 int Add( int a, int b )
    int nSum = Add(2,3);
    cout<<" 2 + 3 = "<<nSum<<endl;

    // 因为参数是浮点型数
    // 调用 float Add( float a, float b )
    float fSum = Add(2.5f,3.7f);
    cout<<" 2.5 + 3.7 = "<<fSum<<endl;
}
```

```
    return 0;  
}
```

经过以上这样的函数重载，编译器会根据不同的数据类型调用相应的重载函数，这样就可以得到正确的结果了：

```
int Add( int a, int b )被调用!  
2 + 3 = 5  
float Add( float a, float b )被调用!  
2.5 + 3.7 = 6.2
```

在这段程序中，根据计算的数据类型的不同，我们对 Add() 函数进行了重载，分别实现了其整型数版本和浮点型数版本。在输出结果中，我们可以清楚地看到第一次对 Add() 函数的调用执行的是整型数版本“int Add(int a, int b)”，而第二次执行的是浮点型数版本“float Add(float a, float b)”。为什么同样是对 Add() 函数的调用，两次执行的却是不同的函数呢？这是因为主函数在第一次调用 Add() 函数时，输入的参数是两个整数，根据参数的类型，编译器找到了第一个整型数版本的 Add() 函数跟它匹配，所以执行第一个 Add() 函数；而在第二次用两个浮点型数做参数调用 Add() 函数时，编译器则会找到第二个浮点型数版本的 Add() 函数跟它匹配，所以会执行第二个 Add() 函数，如图 5-8 所示。

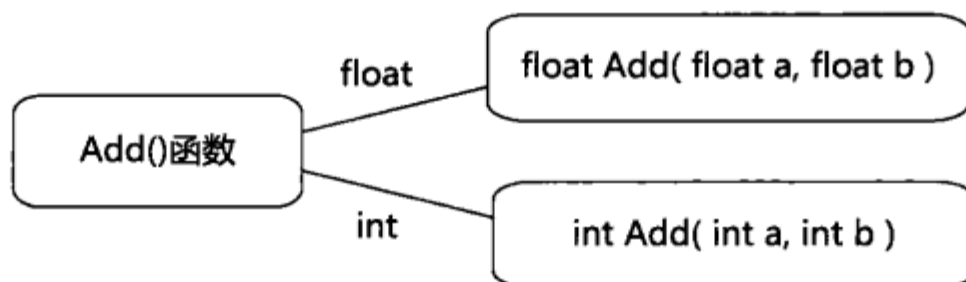


图 5-8 同样的函数名，不同的函数实现

函数重载的意义在于可以根据输入参数的类型或者个数，自动地在多个重载函数中查找与之相匹配的重载函数，从而智能地决定采用哪个函数版本。这样就可以利用函数重载以统一的函数名来实现那些相似的功能，在调用的时候做到形式统一。这些重载函数因为实现的功能相似，所以具有相同的函数名，但是因为它们接受的输入参数不同，所以又有不同的具体实现。

那么，什么时候需要使用函数重载呢？只要发现程序中有多个函数的意义相似，只是处理的数据不同，就可以使用函数重载。简单地讲，只要表达的动作相同，只是动作的对象不同，就可以使用相同的函数名来表示动作，用不同的参数来表示动作对象，这就是重载函数的应用规则了。需要注意的是，我们不要将不同意义的函数定义为重载函数，以免出现对函数调用的误解，这样做也违背了函数重载的本意。

5.3.2 重载函数的解析

我们知道，编译器是通过函数的参数类型和个数来区分重载函数的不同版本的。所以，相同函数名的多个函数，只有相互之间的参数类型或者个数不同，才可以构成合法的重载函数。例如：

```
// 参数类型不同构成合法的函数重载
int max(int a, int b);
float max(float a, int b);
double max(double a, double b);
```

以上三个函数分别接受 int、float 和 double 类型的参数，具有不同的参数类型，因此是正确的函数重载。但要特别注意的是，如果两个函数仅仅是返回值类型不同，并不能构成函数重载，例如：

```
// 仅仅是函数返回值不同，不能构成合法的函数重载
int max(int a, int b);
float max(int a, int b);
```

这是因为函数调用时，函数的返回值有时并不是必需的，这时编译器就无法根据返回值判断到底该调用哪一个重载函数了。

当定义了正确的重载函数后，在调用这些重载函数时，编译器就该忙活了，它需要根据我们在调用时给出的参数找到正确的重载函数。

首先，编译器会进行严格的参数匹配，包括参数个数和类型。如果编译器发现某个重载函数的参数类型和个数都跟函数调用相匹配，则优先调用这个重载函数。例如：

```
int nMax = max(1, 2);
```

根据编译器的重载函数匹配优先顺序，这里调用的应该是“int max(int a, int b)”。因为 1 和 2 这两个参数都是 int 类型，并且参数个数也是两个，跟这个重载函数严格匹配。但是，如果找不到严格匹配的重载函数，编译器会尝试通过内部转换寻求一个匹配，比如把带小数的常数转化为双精度的浮点数再进行匹配。例如：

```
double fMax = max( 1.0, 2.0 );
```

面对这个对重载函数的调用，大家可能有点迷糊了：到底是调用 float 版本的 max() 函数还是 double 版本的 max() 函数呢？

编译器是一个谨小慎微的家伙，它总是害怕丢失数据的精度。所以，一个带小数的常数，例如这里的 1.0，在编译器里默认会被转换为比较保险的 double 类型。这样就比较明显了，这里调用的就是“double max(double a, double b)”。

除了理解重载函数的匹配原则之外，在使用重载函数时，还要特别注意带参数默认值的

函数可能给函数重载带来的错误。例如，可以声明这样两个重载函数：

```
// 比较两个整型数的大小
int max(int a, int b);
// 比较三个整型数的大小，第三个数的默认值是 0
int max(int a, int b, int c = 0);
```

当以“`int nMax = max(1, 2);`”的形式尝试调用这两个重载函数时，以上代码中第二个 `max()` 函数使用了默认参数，这就使得这个调用跟两个重载函数都能匹配，这时编译器就不知道到底该调用哪个函数了。所以在重载函数中应该尽量避免使用默认参数，让编译器能够准确无误地找到匹配的重载函数。

5.4 函数设计的基本规则

函数是 C++ 程序的基本功能单元，就像人体中的细胞，其重要性不言而喻。我们在大量使用别人设计好的函数的同时，也在设计大量的函数供自己或他人使用。函数能否设计好，成为评价开发者水平高低的重要标准。关于函数的设计，业界已经积累了相当多的经验规则。这些经验规则是每个新入行的开发者都应当了解和遵循的，并且需要在开发活动中将这些规则加以灵活应用。函数的两个基本要素是函数的声明和函数的定义，下面分别从这两方面来谈一谈相关的最佳实践经验。

5.4.1 函数声明的设计规则

函数的声明，也称为函数的接口，它是函数跟外界打交道的通道。它就像函数箱子上的标签一样，可通过该标签了解箱子中封装的是什么功能，需要什么样的输入数据，以及能够返回什么样的结果。大量实践表明，一个函数是否好用，往往由其接口设计的好坏决定。在设计实现函数时，不仅要让函数的功能正确，还要让函数的接口清晰明了，有较高的可读性。只有这样，在使用这个函数时，才会清楚函数的功能及函数的输入/输出参数等，从而正确使用这个函数。如果函数的接口不清楚，则很容易造成函数的误用。

在函数接口的设计上，通常有如下几种规则。

1. 使用动词+名词的形式给函数命名

函数是对功能的封装，而功能往往表现为动作和相应的作用对象。比如烧水、擦玻璃、拷贝字符串等，都是某个动作和相应的作用对象组合在一起构成的一个完整功能。所以在给函数命名时，最好使用函数的主要动作和作用对象组合而成的动宾短语，这样能让函数的功能一目了然。例如：

```
// 计算面积
// Get 是动作, Area 是动作的对象
int GetArea();
// 拷贝字符串
// Copy 是动作, String 是动作的对象
char* CopyString(char *strDest, const char *strSrc);
```

2. 使用完整清晰的形式参数名, 表达参数的含义

参数表示函数所作用的对象及需要处理的数据, 也就是函数所表示的动作的宾语。所以, 最好能够使用完整清晰的参数名来明确这个宾语的具体意义。如果某个函数没有参数, 也最好使用 void 填充, 表示这个函数不需要参数。同样是函数, 不同的形式参数名可以带来不同的效果, 例如:

```
// 参数的含义明确,
// 可以清楚地知道第一个参数表示宽度, 第二个参数表示高度,
// 整个函数的功能就是设置宽度和高度的值
void SetValue(int nWidth, int nHeight);
// 参数的含义不明确,
// 只能去猜测这两个参数的含义, 从而也就无法理解整个函数的功能
void SetValue(int a, int b);
// 好的接口设计, 没有参数就使用 void 填充
int GetWidth(void);
// 不好的接口设计
int GetValue();
```

3. 参数的顺序要合理

在某些情况下, 表示特定意义的参数的顺序已经具有了业界普遍遵循的规则, 比如复制字符串函数, 把目标字符串作为第一个参数, 把源字符串作为第二个参数。这些规则我们应当逐渐熟悉并遵守, 而不应该标新立异。例如, 写一个设置矩形参数的函数:

```
// 不遵循参数顺序规则的接口设计
void SetRect( int nRight, int nBottom, int nTop, int nLeft);
```

SetRect()函数接口虽然能够完成传递参数的功能, 但并不是一个好的接口设计, 因为其中参数的顺序不符合业界的规则。如果该函数写好了让其他人使用, 而其他人是按照业界的普遍规则来调用该函数的, 则很可能因为参数顺序的问题而导致函数被错误使用。这里, 规范参数顺序应该是:

```
// 规范的接口顺序
void SetRect(int nLeft, int nTop, int nRight, int nBottom);
```

4. 避免函数有太多的参数

函数的参数个数不宜过多, 应该尽量控制在 5 个以内。如果参数太多, 在使用时容易将参数类型或顺序搞错, 给函数的使用带来困难。如果确实需要传递很多数据, 可以使用一个

结构体来封装所有参数，然后传递整个结构体。例如：

```
// 创建字体函数  
HFONT CreateFontIndirect( const LOGFONT *lplf );
```

这里，使用了 LOGFONT 结构体来定义字体的多个属性，通过传递一个结构体指针向 CreateFontIndirect() 函数传递关于字体的多个参数。

5. 使用合适的返回值

有时候函数原本不需要返回值，但为了增加函数的灵活性，往往给函数附加一个返回值，用来表示函数执行成功与否。如果函数返回值用于出错处理，这样的返回值一定要清楚、准确，采用一些比较特殊的值，比如 0、-1 或 NULL 等，也可以是自己定义的错误类型编号等。

好的函数遵循的规则可以用图 5-9 来表述。

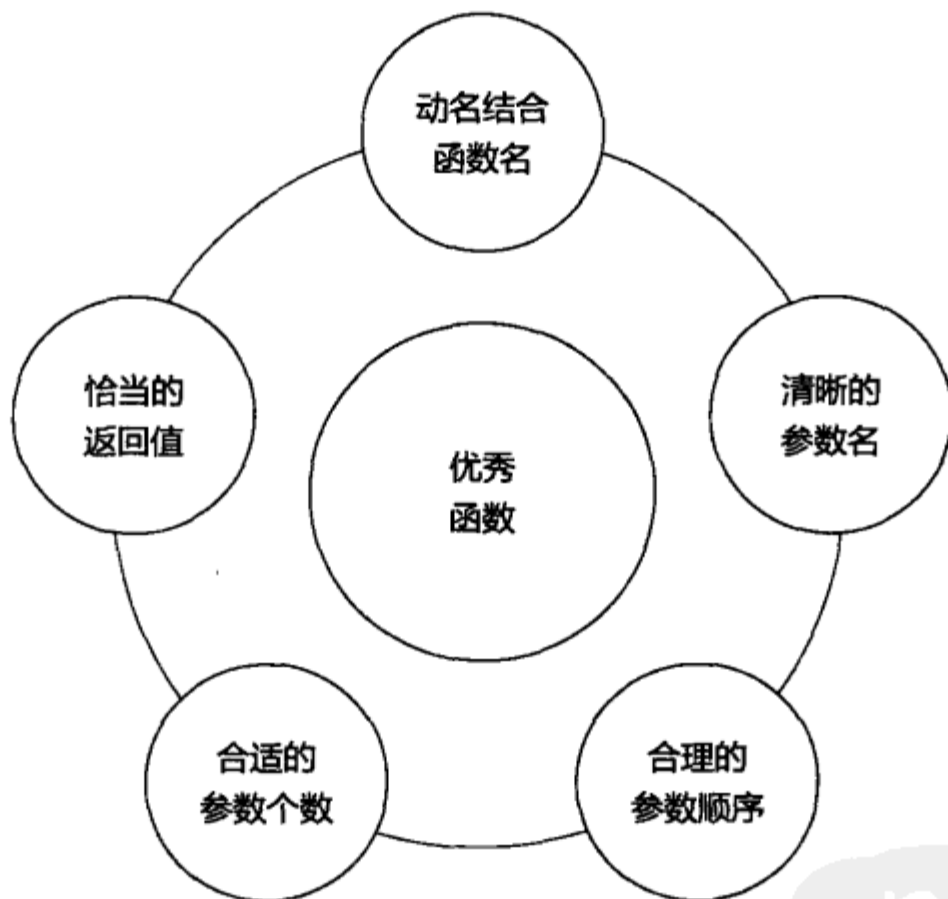


图 5-9 优秀函数的五项修炼

5.4.2 函数体的设计规则

设计好函数箱子的标签后，我们就能更轻松地决定该使用哪一个函数及如何使用该函数。至于这个函数是否能用，就要靠我们对函数主体的设计、靠我们的内功了。

虽然各个函数实现的功能不同，函数主体也大不一样，但还是有一些普遍适用的经验规则供我们学习和参考，从而设计出优秀的函数体。

1. 在函数体的“入口处”，对参数的有效性进行检查

某些函数对参数是有特定要求的，例如，设置年龄的函数，其参数当然不能为负数，函数的指针参数也不能为 NULL。面对这样的情况，就需要在函数的“入口处”对函数参数的有效性进行检查，以提高函数的正确性。

如果需要对无效的参数进行处理，则可以采用条件语句，根据参数的有效性对用户进行提示或者直接返回函数执行失败信息等。例如：

```
// 设置年龄
bool SetAge( int nAge )
{
    // 在函数入口处对参数的有效性进行检查
    // 如果参数不合法，则提示用户重新设置
    if( nAge < 0 )
    {
        cout<<"设置的年龄不能为负数，请重新设置。"<<endl;
        // 返回函数执行失败
        return false;
    }
    // 如果参数合法，则继续进行处理...
}
```

这里，我们首先在函数入口处对参数的有效性进行了检查。如果参数不合法，则提示用户重新进行设置，并返回一个表示函数执行失败的返回值；如果参数合法，就继续进行处理。通过对参数的有效性进行检查，可以很大程度上提高函数功能的正确性，防止错误参数的处理。

如果无须处理无效的参数，还可以简单地使用断言 (assert) 来对参数的有效性进行判断，防止函数被错误地调用。断言可以接受表达式为参数，如果整个表达式的值为 true，则断言不起作用，函数继续执行。如果表达式的值为 false，断言就会提示我们断言条件不成立，函数的参数不合法，需要我们进行处理。例如，要设计一个除法函数 Divide()，在该函数中，我们要对除数为 0 的情况进行防错处理，因此可以使用断言来进行判断并提示函数是否被错误地调用。

```
#include "stdafx.h"
#include <assert.h>    // 引入断言头文件
using namespace std;

double Divide( int nDividend, int nDivisor )
{
    // 使用断言判断除数是否为 0，进行防错处理
    assert( 0 != nDivisor );

    return (double)nDividend/nDivisor;
}
```



```
int _tmain(int argc, _TCHAR* argv[])
{
    // 除数为 0, Divide()函数被错误地调用
    double fRet = Divide( 3, 0 );

    return 0;
}
```

在主函数中以 0 为除数调用 Divide()函数，当函数执行到断言时，断言会判断条件是否为 true，如果为 true，则继续执行；如果为 false，则系统会弹出调试错误提示框，提醒我们函数的参数不合法，函数被错误地调用了。

2. 谨慎处理函数返回值

如果函数有返回值，则不可返回一个指向函数体内部声明的局部对象的“指针”或者“引用”，因为这些局部对象会在函数执行结束时被自动销毁，这些指针或引用所指向的数据就变得无效，成为“野指针”。当我们在函数返回后再次尝试使用这些“指针”时，其内容可能已经被修改，不再是从函数内部返回的数据，而这些不确定内容的“指针”，会给程序带来很大的安全隐患。例如：

```
int* GetVal()
{
    int nVal = 5;
    // 返回一个局部变量 nVal 的指针是极危险的
    return &nVal;
}
```

当在函数之外得到这个指针并继续使用时，什么事情都可能发生，例如：

```
// 得到一个从函数返回的指向其局部变量的指针
int* pVal = GetVal();
// 没人会预料这个动作会产生什么样的结果，也许地球会因此毁灭
*pVal = 0;
```

3. 函数的功能要单一

一个函数的功能要单一，不能让函数具有多个功能，如果一个函数需要完成多项任务，最好拆分成多个函数分别完成。

4. 函数主体不宜太长

女孩们追求苗条身体，函数也不例外。函数主体的长度不宜太长，尽量控制在 100 行代码之内。如果发现你的函数太长，可能已经违反了上一条“函数的功能要单一”的规则，整个函数实现了过多的功能，则需要拆分成多个函数以达到瘦身的目的。

当 C++ 爱上面向对象

很多第一次进入 C++ 世界的人都会问：C++ 中的两个加号到底是什么意思？

C++ 语言是从 C 语言发展起来的，C++ 比 C 多出的两个加号，实际上是 C++ 的自增运算符，表示 C++ 语言是在 C 语言的基础上添加了新的内容。如果其中一个加号代表 C++ 在 C 的基础上增加了模板、异常处理等现代程序设计语言的新特性，那么另外一个加号则代表 C++ 支持面向对象程序设计思想。正是这两个加号让 C++ 语言与 C 语言有了本质的区别。其中的面向对象程序设计思想完成了从 C 语言到 C++ 语言的进化，从而让 C++ 语言既具备了 C 语言的优秀“根基”，又能够代表现代程序设计语言的发展趋势，使得 C++ 在多种程序设计语言中经久不衰。

武林中流传这样一句话：“平生不识陈近南，纵称英雄也枉然！”这句话说明了陈近南的名声之响。如果陈近南是武林中响当当的人物，那么面向对象程序设计思想则是程序设计界名副其实的老大。可以毫不夸张地说，作为一名程序员，如果不知道面向对象程序设计思想，纵称高手也枉然。

既然面向对象程序设计思想的名声如此大，那么它到底是怎么回事？它跟 C++ 语言有怎样的爱恨情仇？它为何会受到这么多人的追捧和欢迎？别着急，且听我一一道来。

6.1 从结构化设计到面向对象程序设计

面向对象程序设计的雏形早在 1960 年的 Simula 语言中出现过，当时程序设计领域正面临一种危机：在软硬件系统逐渐复杂的情况下，软件如何得到良好的设计和维护？面对越来越复杂的软硬件系统，传统的 C 语言风格的结构化设计思想已越来越满足不了现实的需要——结构化设计无法很好地描述整个系统，也让人难以理解，因而给软件后期的维护增加了难度，人们正陷入一场前所未有的软件危机中。为了解决这场软件危机，人们开始寻找解决这场软件危机的“银弹（silver bullet）”。而面向对象程序设计思想正是在这种背景下，通过强调可重复性解决了这一问题。20 世纪 70 年代的 Smalltalk 语言在面向对象方面堪称经典——以至于 40 年后的今天依然将这一语言视为面向对象语言的“老祖宗”。

《人月神话》是软件领域里一本具有深远影响力的著作。它诞生于软件危机之下，提出了“10 年之内无银弹”，即没有任何一项技术或方法可以让软件工程的生产力在 10 年内增长 10 倍。

在西方基督教的传说中，只有银弹击中心脏，才可以杀死怪物。而在这本书中，作者把规模越来越大的软件开发项目比作无法控制的怪物，希望有一项技术能够像银弹杀死怪物那样，以彻底解决这个问题。

虽然面向对象程序设计思想并不是银弹，也不能够解决所有大型软件项目的问题，但是它提出了一种描述软件的更加自然的方式，在一定程度上解决了这场软件危机。

6.1.1 “自顶向下，逐步求精”的结构化程序设计

在学习新的面向对象程序设计思想之前，先来看看在面向对象程序设计思想出现之前的软件是如何设计和开发的。回顾前面章节中曾经学习过的例子：先提出问题；然后分析问题的处理流程；接着根据需要把一个大问题划分为几个小问题，分成各个子模块，解决每个小问题，实现每个子模块；最后通过主函数按照某种次序调用这些子模块，组织业务逻辑流程，最终解决问题，如图 6-1 所示。像这样从问题出发，自顶向下、逐步求精的开发方法称为“结构化程序设计方法”。

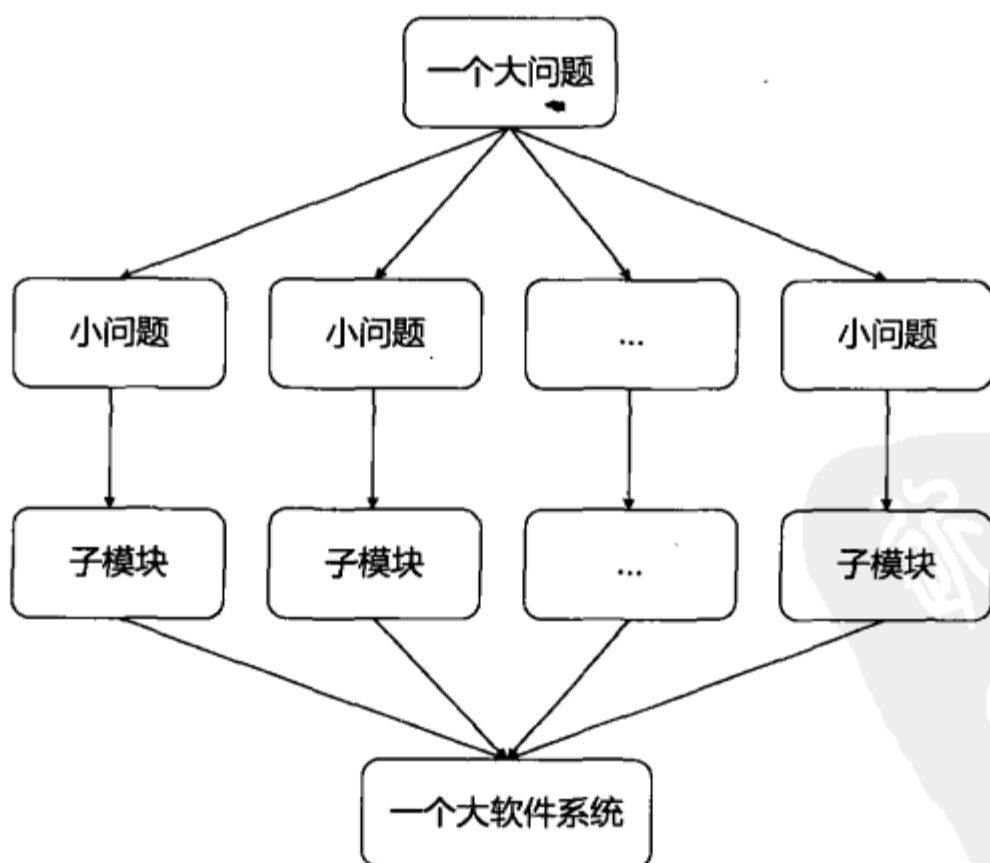


图 6-1 结构化程序设计的流程

结构化程序设计思想诞生于 20 世纪 60 年代，鼎盛于 20 世纪 80 年代，成为当时最为流

行的程序设计思想。结构化程序设计思想的流行有其内在原因，跟当时其他程序设计思想相比，结构化程序设计思想有着明显的优势。

1. 程序仅由三种基本结构组成

正如第 4 章中所介绍的程序流程控制结构一样，结构化程序设计思想限定任何复杂的程序只有三种基本结构，分别为顺序结构、选择结构和循环结构。任何程序逻辑都可以用这三种基本的结构经不同的组合或嵌套来实现。这样使得程序的结构相对比较简单，易于实现和维护。

2. 分而治之，各个击破

人们在解决复杂问题时，总是采用“分而治之”的策略，把大问题分解为多个小问题后，再“各个击破”。结构化程序设计采用相同的策略，把较大的程序划分为多个子模块，再把子模块按层次关系进行组织，最后分工逐个完成这些子模块。按照一定的原则，把大问题分割为小问题“各个击破”，符合人们思考问题的一般规律。通过分解问题，降低了问题的复杂度，使得程序易于编写和维护。同时，分解后的小问题（子模块）可由多人分工协作完成，提高了开发效率。

3. 自顶向下，逐步求精

结构化程序设计思想倡导的方法是“自顶向下，逐步求精”。所谓“自顶向下，逐步求精”，就是先从宏观角度考虑，按照功能或者业务逻辑划分程序的子模块，定义程序的整体结构，再对各个子模块逐步细化，最终分解到程序语句为止。这种方法使得程序设计者能够全面考虑问题，使程序的逻辑关系清晰明了，使整个程序设计开发的过程变成考虑“先做什么，再做什么”而不是具体“怎么做”的问题。这样程序的实现变得更加简单了。

虽然结构化程序设计思想有许多优点，但是随着时代的发展，软件需要解决的问题也越来越复杂。在利用结构化程序设计思想解决复杂问题的时候，缺点也逐渐暴露出来：在结构化程序设计中，数据和操作是相互分离的，这就导致如果数据的格式发生变化，相应的操作函数就要改写；如果遇到系统需要扩展功能，还涉及模块的重新划分，要修改大量原先写好的功能函数。结构化程序设计中数据和操作相互分离的特点使得一些模块跟具体的应用环境结合紧密，旧有的程序模块很难在新的程序中得到复用。这些结构化程序设计思想的固有缺点是它越来越不适合大型的软件项目的开发，所以人们开始寻找一种新的程序设计思想。正是在这种情况下，一些新的程序设计思想开始逐渐取代结构化程序设计思想，而面向对象程序设计思想就是其中的“带头大哥”。

6.1.2 面向对象程序设计

面向对象程序设计（Object Oriented Programming, OOP）是对结构化程序设计的继承和

发展，它不仅汲取了结构化程序设计的精华，而且以一种更接近人类思维的方式来分析和解决问题：程序是对现实世界的描述，现实世界的基本单元是物体，程序中的基本单元就是对象。

面向对象程序设计思想认为：现实世界是由很多彼此相关并互通信息的实体——对象（object）组成的。大到一个星球、一个国家，小到一个人、一个分子，无论是有生命的，还是没有生命的，都可以看成是一个对象。通过分析这些对象，发现每个对象都由两部分组成：描述对象状态或属性的数据和描述对象行为或功能的函数（方法）。结构化程序设计思想将数据和函数分开，而面向对象程序设计将数据和函数紧密结合共同构成对象来更加准确地描述现实世界。这可以说是两者最本质的区别。

跟现实世界相对应的，在面向对象程序设计思想中，我们用对象来代表现实世界中的实体，每个对象都有自己的属性和行为，而整个程序由一系列相互作用的对象构成，对象之间通过互相发送消息来完成互通信息，以此完成复杂的业务逻辑。比如在一个学校中，可以利用面向对象思想将老师和学生这两种实体抽象成对象，这样其中的有部分属性是相同的，比如姓名、年龄等；而有部分属性是不同的，比如老师有职务这个属性，而学生则没有。另外，老师和学生这两种对象还有各自不同的行为；比如老师有备课、上课、批改作业的行为，而学生则有听课、完成作业等行为。老师和学生各自负责自己的行为 and 职责，同时又相互发生联系，老师上课、学生听课。通过对象之间的相互作用，学校才能正常运作。整个流程如图 6-2 所示。

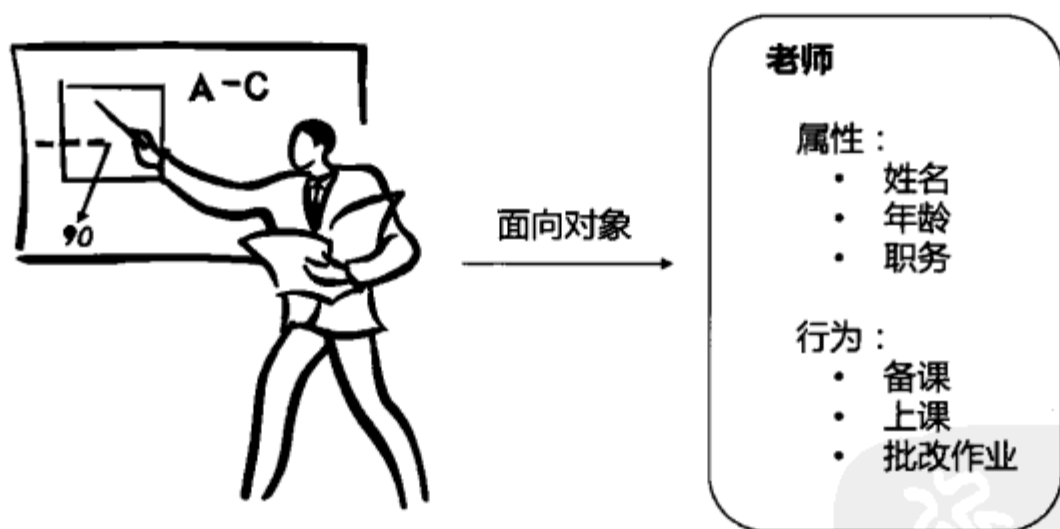


图 6-2 用面向对象思想将老师抽象成对象

6.1.3 面向对象的三座基石：封装、继承与多态

面向对象程序设计思想是在传统的结构化程序设计思想基础上发展起来的。那么是哪些特点让面向对象程序设计思想从根本上区别于结构化程序设计思想，使其成为一种全新的程序设计思想呢？通常认为，封装、继承与多态是面向对象思想的三座基石，它们共同构成了面向对象思想的核心特征，如图 6-3 所示。

1. 封装

程序是用来描述现实世界的，那么在现实世界中又是如何描述周围的物体的呢？我们总说某个物体是什么、能做什么，这就是描述物体所必需的数据和算法。

在传统的结构化程序设计思想中，程序中的数据和算法是相互分离的。也就是说，在描述一个物体的时候，物体是什么（数据）和物体能做什么（算法）是相互分离的。但是在面向对象思想中，它是通过封装机制将数据和相应的算法捆绑到一起的，以形成一个完整的、具有属性（数

据）和行为（算法）的对象，避免了外界的干扰和不确定性。简单来说，对象就是封装数据和操作这些数据的算法的逻辑实体，也是现实世界中物体在程序中的反映，如图 6-4 所示。

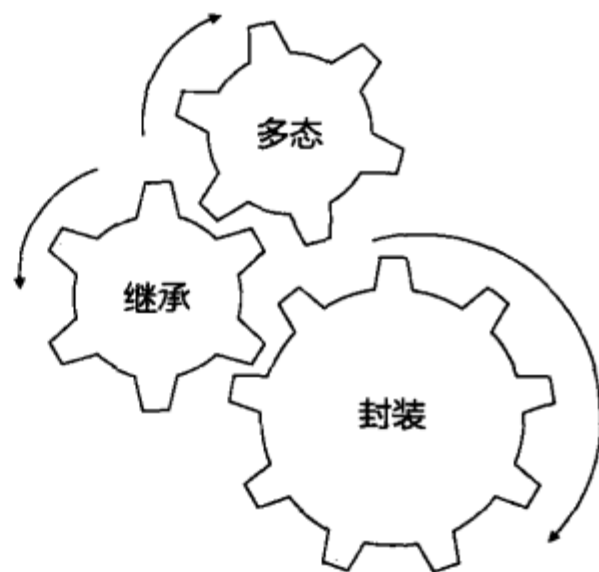


图 6-3 面向对象思想的三座基石

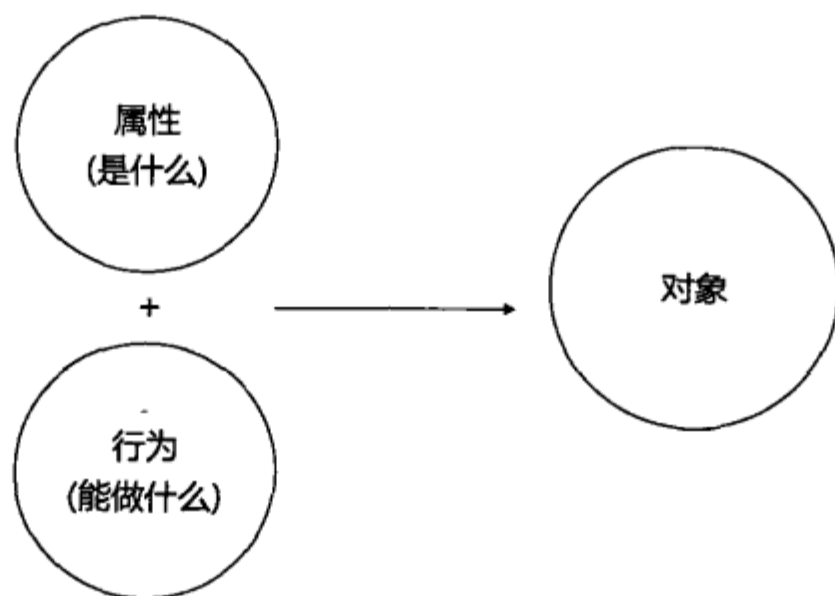


图 6-4 将数据和算法封装成对象

封装机制还带来了另外一个好处，那就是对数据的保护。在结构化程序设计中，因为数据和算法是相互分离的，某些算法可能错误地修改了不属于它的数据。在对象内部，某些算法和（或）某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关部分意外地改变或错误地使用了对象的私有部分。如同钱包里的钱是我们的私有财产，别人是不可以访问的，当然，小偷除外。

2. 继承

继承是可以让某个类型的对象获得另一个类型的对象的属性的方法。如同现实世界中的等级进化一样，语文老师属于老师这个类别，具有了老师这个类别的共有属性；而老师又属

于人类这个类别，具有了人类的共有属性。这种继承的原则可以让每个子类都轻松具有父类的公共特性。

正是这种从父类继承属性的特点，可以很好地支持代码的重用。比如，想给一个已有的类别增加新的属性，而又不想改变这个类别。继承就可以从这个已存在的类别派生一个新的类别来实现。这个新的类别将具有原来类别的特性和新添加的特性。而继承机制的强大魅力就在于它允许我们充分利用已经存在的类别（接近需要，而不是完全符合需要的类），同时又可以以某种方式修改这个类别，添加新的特性，而不会影响其他的东西，如图 6-5 所示。

3. 多态

“见人说人话，见鬼说鬼话”，是说一个人两面三刀，不是什么好人。可在 C++ 世界中，这种在不同情况下做不同事情的现象，却冠以一个冠冕堂皇的名字——多态，而成为面向对象思想的一个重要特征。

多态，就是指对象在不同情况下具有不同形式的能力。对于不同的对象，某个操作可能会有不同的行为，这依赖于这个动作所要操作数据的类型。比如说求和操作，如果操作的数据是数，则它是对两个数求和；如果操作的数据是两个字符串，则它是连接两个字符串。这就是 C++ 世界中的“见人说人话，见鬼说鬼话”。

多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（操作）可以相同的方式予以调用。简单来说，多态机制允许通过相同的接口引发一组相关但不相同的动作，通过这种方式可以减少代码的复杂度。在某种特定的情况下应该做出怎样的动作由编译器决定，而不需要程序员手工进行干预，为程序员省了很多事，如图 6-6 所示。

纵观面向对象思想的三大特征，它们是紧密相连、不可分割的。通过封装，可以将现实世界中的事物描述为对象，将对象的数据和对象的算法封装到一起，实现数据的隐藏；通过继承，可以体现类与类之间的关系，实现设计和代码的复用；同时继承可以带来对象的多态性，从而完整地构成了面向对象的三大基本特征。

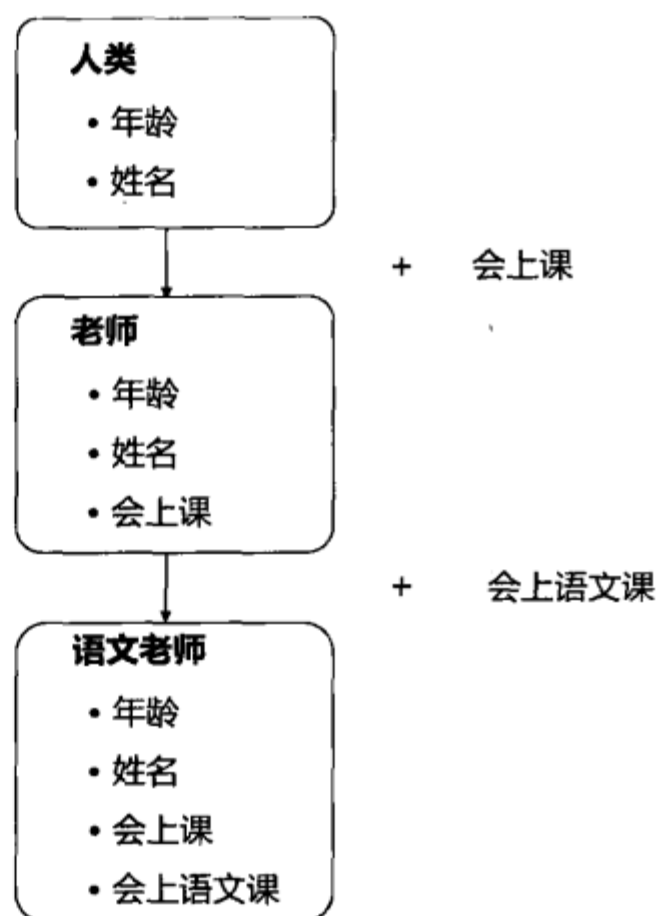


图 6-5 继承

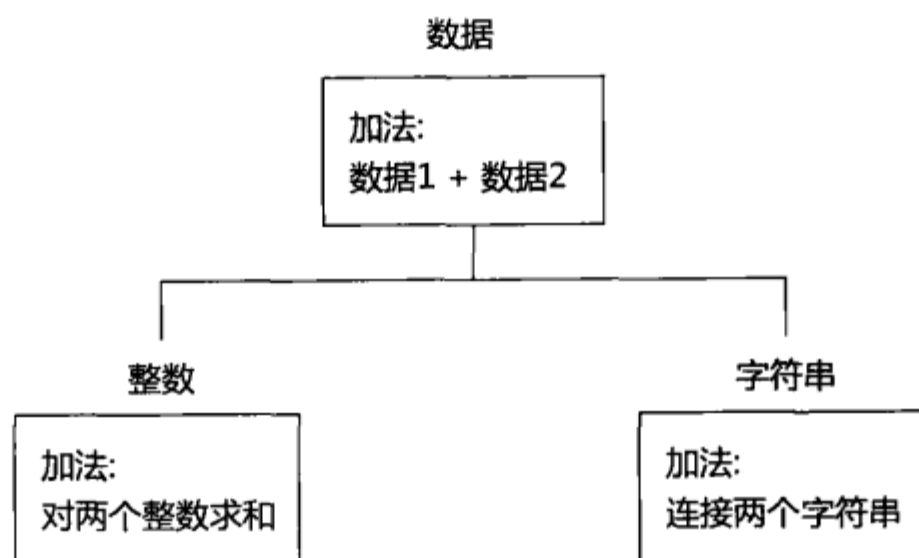


图 6-6 多态

正是因为面向对象思想的封装、继承和多态这三大特性，使得面向对象思想在程序设计中具有不可替代的优势。

(1) 容易设计和维护。

面向对象思想强调从客观存在的事物（对象）出发来认识问题和解决问题，因为这种方式更加符合我们认识事物的规律，大大降低了问题的理解难度。面向对象思想所运用的抽象、封装、继承等基本原则，符合人类日常的思维方法和原则，使得采用面向对象思想设计的程序结构清晰、可读性高。由于继承的存在，即使需求改变，也只在局部模块进行维护，所以维护起来非常方便，所耗成本也较低。

(2) 复用设计和代码，系统质量高。

面向对象思想的继承和多态，强调了程序设计和代码的重用，这在设计新系统的时候，可以最大限度地重用已有的、经过大量实践检验的设计和代码，使系统能够满足业务的需求并具有较高的质量。同时，因为复用了以前的设计和代码，大大提高了开发效率。

(3) 容易扩展。

开发大型系统的时候，最担心的就是需求的变更、对系统进行扩展、利用面向对象思想继承、封装和多态的特性，可以设计出高内聚、低耦合的系统结构，可让系统更灵活、更易扩展，从而轻松应对系统的扩展需求，降低成本。

面向对象程序设计思想在软件开发中的这些优势成为当前最为流行的程序设计思想之一，是每个进入 C++ 世界的程序员都需要理解和掌握的。它就像程序设计中的《易筋经》般博大精深，而这里所介绍的只是面向对象思想最基础和最入门的知识，要完全领会和灵活运用面向对象思想，还需要在实践中不断学习和总结，只有这样才能增加对软件设计和开发的功力，成为真正的高手。

对面向对象程序设计思想的支持，是 C++ 语言的精华所在。还记得 C++ 最开始的名字吗？“C with class”，也就是说，C++ 语言是因为面向对象思想而区别于 C 语言的，面向对象思想是两者最本质的区别。

作为程序设计的初学者，要在一开始就完全理解可能有一定的难度，所以在学习本章的时候，不仅要着重概念的理解，还要通过示例的演练体会 C++ 是如何体现面向对象思想的，以及如何利用面向对象思想构建更加灵活的 C++ 程序。只有理论联系实际，才能深刻体会到面向对象程序设计思想的精髓，从而在以后的开发活动中加以应用。

6.2 类：当 C++ 爱上面向对象

窈窕淑女，君子好逑。面向对象思想一出现就受到众人的追捧，当然这其中也包括 C++。当 C++ 爱上面向对象时，于是就诞生了“类”。

6.2.1 类的声明和定义

类这个概念是 C++ 和面向对象思想相结合的结晶，正是通过它，面向对象思想才得以在 C++ 中得到完美体现。

面向对象思想把现实世界中的物体都封装成对象，而类是所有相同类型对象的抽象，是它们总体的描述。比如，学校有很多老师，每个老师各不相同，但是作为老师这类对象的抽象，老师这个类却是相同的，因为有相同的属性和行为。所以，在一个类的声明中描述这类对象的属性和行为，从而用这个类描述这一类对象。在 C++ 中，声明一个类的语法格式如下：

```
class 类名 : public 基类名
{
public:
    // 公有成员，通常用来定义类的行为，提供接口供外部访问
protected:
    // 保护型成员
private:
    // 私有成员，通常用来定义类的属性
};
```

其中，class 是 C++ 中声明类的关键字，其后跟类的名字，通常用一个名词来描述这个类所代表的一类对象，再后面是类的继承方式和基类的名字。如果没有基类，则这一部分可以省略。

完成类的名字及继承关系的定义后，可以开始在类的主体中描述这个类的属性和行为。我们知道，类实际上是对多个同类型对象的抽象。通过面向对象的封装机制，可将现实世界

中的物体封装成对象，这个对象就包含了这个物体的属性和行为。在 C++ 中，我们总是用一些变量来代表某些数据，将变量引入类的声明之中成为类的成员变量，这些变量就成为对对象属性的描述。比如，一个老师的年龄，可以在类中添加一个整型变量来描述；他的姓名，可以在类中使用一个字符串变量来描述。通过这些成员变量，可以描述一个对象的所有属性。

除了对象的属性之外，对象的另外一个重要组成部分就是它的行为。在 C++ 中，我们用函数来描述一个行为动作。同样，我们也将函数引入类中成为它的成员函数，用来描述类对象的行为。比如，一个老师有备课的行为动作，我们就可以为老师这个类添加一个 `PrepareLesson()` 函数，在这个函数中可以对老师备课动作进行具体的定义。类的构成如图 6-7 所示。

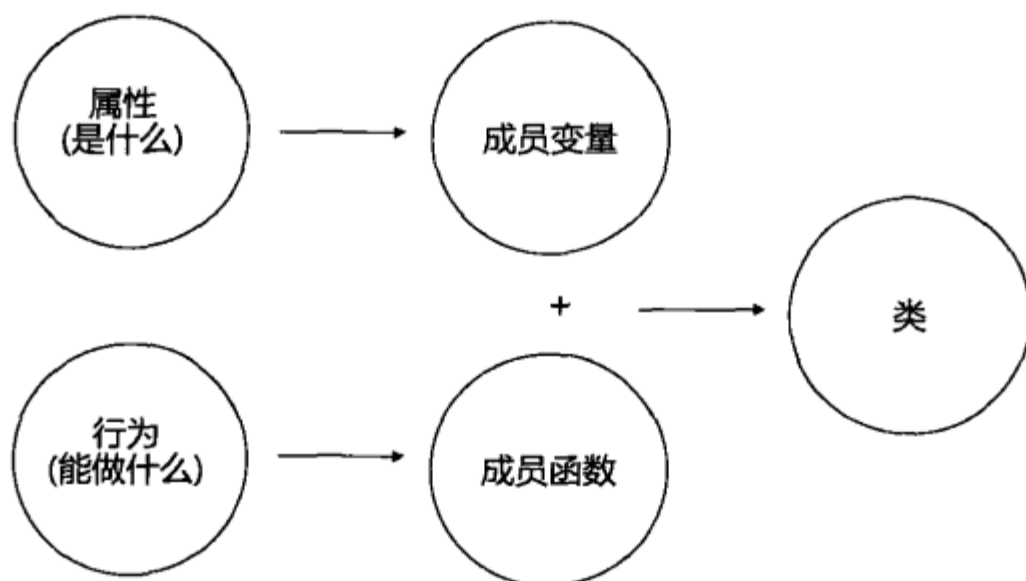


图 6-7 类的构成

最佳实践：为类设计友好的接口

所设计的类不仅供我们自己使用，更多时候它还会提供给其他程序员使用，以达到代码复用的目的。这时，类的接口设计的好坏，将会影响到他人能否正确并轻松地使用我们所设计的类。类的接口设计的友好与否，成为衡量一个程序员水平的重要标准。

类的接口，就像类的说明书一样，是向类的使用者说明它所需要的资源及它能够提供的服务等。只要类的接口设计得友好，就可以轻松地知道如何使用这个类。要让类的接口友好，应当遵守下面的设计原则。

- 简化的视图

接口，就是类所能够提供的服务。所以，在进行类的接口设计的时候，只需要将必要的成员函数公有（`public`）就可以了，使用受保护的（`protected`）或者私有的（`private`）成员来向用户隐藏不必要的细节。这样不仅可以达到封装的目的，还可以减少用户的错误率。

- 用户的词汇

类设计出来最终是让用户使用的，所以在设计类的接口的时候，应该考虑从用户的角度出发，让用户在阅读类的接口的时候，不需要学习新的词汇或概念，这样可以降低用户的学习曲线。

在类的主体中，可以使用 public、protected 及 private 三个关键字来定义类的属性和行为的访问级别。通常，会在 public 部分定义类的行为，提供公共的函数接口供外部访问；在 protected 部分，可以定义遗传给下一代子类的属性和行为；最后在 private 部分，可以定义这个类所私有的属性和行为。关于类的继承方式和访问控制，稍后将进行详细介绍。这里先来看一个实际的例子。例如，要定义一个类来描述老师这一类对象，通过面向对象思想的封装机制，我们简单地将老师这类对象看成是拥有姓名属性和备课行为的对象。当然，老师还有很多其他属性和行为，这里根据需要作了简化。于是，就有了老师这个类的声明。

```
class Teacher
{
    // 成员函数
    // 描述对象的行为
public:
    void PrepareLesson();    // 备课
    // 成员变量
    // 描述对象的属性
protected:
    string m_strName;        // 姓名
private:
};
```

通过这段代码，我们声明了一个 Teacher 类，它代表了所有老师这种对象的一个抽象描述。这个类继承自基类 Human，它有一个函数是 PrepareLesson()，它代表老师这类对象的行为——备课，这样的函数称为类的成员函数。它还有一个变量是 m_strName，表示老师这类对象的一个共有属性——姓名，这样的变量称为类的成员变量。这样，通过在一个类中利用函数描述对象的行为，利用变量描述对象的属性，就完整地声明了一个用于描述某类对象的类。

完成类的声明之后，就需要对类的行为进行具体的定义了。可以直接在类的声明部分定义类的行为，也就是定义它的成员函数。

```
class Teacher
{
    // 成员函数
    // 描述对象的行为
public:
    void PrepareLesson()    // 备课
    {
        cout<<"老师备课。"<<endl;
    };
};
```

```
//...  
};
```

更多时候，我们只是将类的声明放在头文件（.h）中，在头文件中声明类成员函数行为的原型，成员函数的具体实现一般放在类的外部定义，也就是相应的源文件（.cpp）中。在这个类对应的源文件中，可以对 Teacher 类的 PrepareLesson() 函数进行定义，完成具体的行为动作。

```
// Teacher 类的定义  
void Teacher::PrepareLesson()  
{  
    cout<<"老师备课。"<<endl;  
}
```

这里，我们在类名和函数名之间加上域操作符“::”来表示 PrepareLesson() 函数是属于 Teacher 类的成员函数。成员函数的定义也跟普通函数一样，可以在这个函数中描述这类对象的行为，例如，这里只是输出一个字符串表示老师备课的动作。当然，在实际应用中还有更加复杂的行为。

在 C++ 中，要声明和定义一个类，除了可以使用“class”关键字之外，还可以使用 3.8 节中介绍的“struct”关键字。在语法上，“class”关键字和“struct”关键字非常相似，两者唯一的区别就是，默认情况下用“class”声明的类成员是私有的（private），而用“struct”声明的类成员是公有的（public）。例如：

```
// 使用“struct”声明一个 Rect 类  
struct Rect  
{  
    // 类的成员函数，默认情况下是公有的（public）  
    int GetArea()  
    {  
        return m_nW * m_nH;  
    }  
    // 类的成员变量，默认情况下也是公有的（public）  
    int m_nW;  
    int m_nH;  
};
```

这里，我们使用“struct”声明并定义了一个 Rect 类，默认情况下，使用“struct”关键字声明的类成员都是公有的，可以直接访问类的成员函数和成员变量。例如：

```
Rect rect;  
rect.m_nH = 3;  
rect.m_nW = 4;  
cout<<"Rect 的面积是："<<rect.GetArea()<<endl;
```

当然，无论是使用“class”还是“struct”声明一个类，我们都应该在声明中明确指出各个成员的访问级别，而不是依赖于关键字的默认行为。

“class”和“struct”除了上面这点在类成员默认访问级别上的差异之外，从感情上讲，大多数程序员感到“class”和“struct”在语义上还有很大的差别。“struct”仅像一堆缺乏封装和功能的开放的内存位；而“class”更像活的并且可靠的现实实体，它可以提供服务、有牢固的封装机制和定义良好的接口。既然大多数人都这么认为，那么只有在类有很少的方法并且有公有数据时，才使用“struct”关键字；否则，使用“class”关键字。

6.2.2 使用类创建对象

完成某个类的声明并且定义成员函数之后，这个类就可以直接使用了，可以用它来创建对象，描述现实世界的各种实体。比如 6.2.1 小节中完成的 Teacher 类的声明和定义，就可以用它来创建一个 Teacher 类的对象，描述某一位老师。要得到类的具体对象，可以用定义变量的方式将类当成一种新的数据类型来创建变量，也就是该类的对象，语法格式如下：

类名 对象名；

其中，类名就是定义好的类的名字，对象名就是要声明的对象的标识符，例如：

```
// 定义一个 Teacher 类型的对象 MrChen
Teacher MrChen;
```

这样就得到了一个 Teacher 类型的对象 MrChen，它代表学校中的某位陈老师。得到对象后，就可以调用这个类提供的公有成员函数，完成这个对象的行为。我们使用“.”操作符来调用一个对象的成员函数，其语法格式如下：

对象名.公有成员函数；

例如，要让刚才定义的对象 MrChen 开始备课，就可以调用它的表示备课行为的成员函数：

```
// 调用对象所属类的成员函数，这位老师开始备课
MrChen.PrepareLesson();
```

这样，该对象就会执行 PrepareLesson() 函数，完成备课行为。

除了直接使用对象变量之外，跟普通的数据类型一样，还可以使用对象类型的指针来指向该对象，通过指针来访问该对象的成员。例如：

```
// 定义一个可以指向 Teacher 类型对象的指针 pMrChen
Teacher* pMrChen = NULL;
// 将该指针指向 MrChen 对象
pMrChen = &MrChen;
```

这里，我们首先定义了一个可以指向 Teacher 类型对象的指针，然后通过取地址运算符“&”取得 MrChen 对象的地址并赋值给该指针，这样就将该指针指向了 MrChen 对象。

除了可以使用“&”取地址运算符取得已有对象的地址，并用它对指针进行赋值之外，还

可以使用“new”关键字直接创建对象并返回该对象的地址，再用这个地址对指针进行赋值，同样可以创建新的对象并将指针指向这个新的对象。例如：

```
// 创建一个新的对象
// 并让 pMrChen 指针指向这个新对象
Teacher* pMrChen = new Teacher();
```

这里，我们使用“new”关键字创建了一个 Teacher 类的对象，“new”关键字会调用 Teacher 类的构造函数完成对象的创建，并返回这个对象的地址，再将这个返回的对象地址赋值给 pMrChen 指针，这样就同时完成了对象的创建和指针的赋值。

有了指向对象的指针，就可以利用“->”运算符（这个运算符是不是很像一个指针？）通过指针访问该对象的成员。例如：

```
// 通过指针访问对象的成员
pMrChen->PrepareLesson();
```

要特别注意的是，跟普通的变量不同，使用“new”关键字创建的对象无法在结束其生命周期的时候自动销毁，所以必须使用“delete”关键字销毁这个对象，释放其占用的内存。例如：

```
// 销毁指针所指向的对象
delete pMrChen;
```

“delete”首先会调用 Teacher 类的析构函数~Teacher()完成这个对象特有的清理工作，然后释放掉这个对象所占用的内存，整个对象也就销毁了。

最佳实践：无须在“new”之后或者“delete”之前测试指针是否为 NULL

很多来自 C++ 的“高手”都会强调，为了增加代码的鲁棒性，我们在使用“new”关键字创建一个对象之后，或者在使用“delete”关键字释放一个对象之前，都需要测试指针是否为 NULL。例如，经常会见到这样枯燥无味的代码：

```
Teacher* p = new Teacher();
if (p == NULL)    // 在使用“new”创建对象之后测试指针是否为 NULL
{
    std::cerr << "无法创建 Teacher 对象" << endl;    // 输出错误信息
    abort();
}
```

实际上，在“new”之后和“delete”之前测试指针 NULL 都是多此一举。一方面，在 C++ 中，如果运行时系统无法为 Teacher 对象分配足够的内存，则会抛出一个 std::bad_alloc 异常，“new”操作永远不会返回 NULL。另一方面，C++ 语言保证，如果 p 等于 NULL，则“delete p”不作任何事情。所以，在使用“new”和“delete”关键字创建或者释放对象的时候，只需要简单就好。

```
// 创建对象
Teacher* p = new Teacher();
//...
```

```
// 释放对象
delete p;
// 释放对象之后，需要将指针赋值为 nullptr，表示这个指针已经释放，不可使用
p = nullptr;
```

也许大家会问：类和对象的关系是怎么样的？为什么要定义一个类后再用这个类来定义对象？在 C++ 中，是用类的概念来反映面向对象思想的。类是一种抽象机制，它描述的是同一类对象所共有的属性和行为。比如，学校里有多位老师，老师这个类描述的是这多位老师，也就是多位老师这类对象所共有的属性和行为。反过来，类的对象就是该类的一个特定实体，比如学校某个老师就是 Teacher 这个类的一个特定实体，也就是这个类的对象。简单来讲，多个对象的抽象是类，类的具体化是对象。多位老师可以抽象成 Teacher 类，而陈老师就是 Teacher 类的一个具体对象了，如图 6-8 所示。

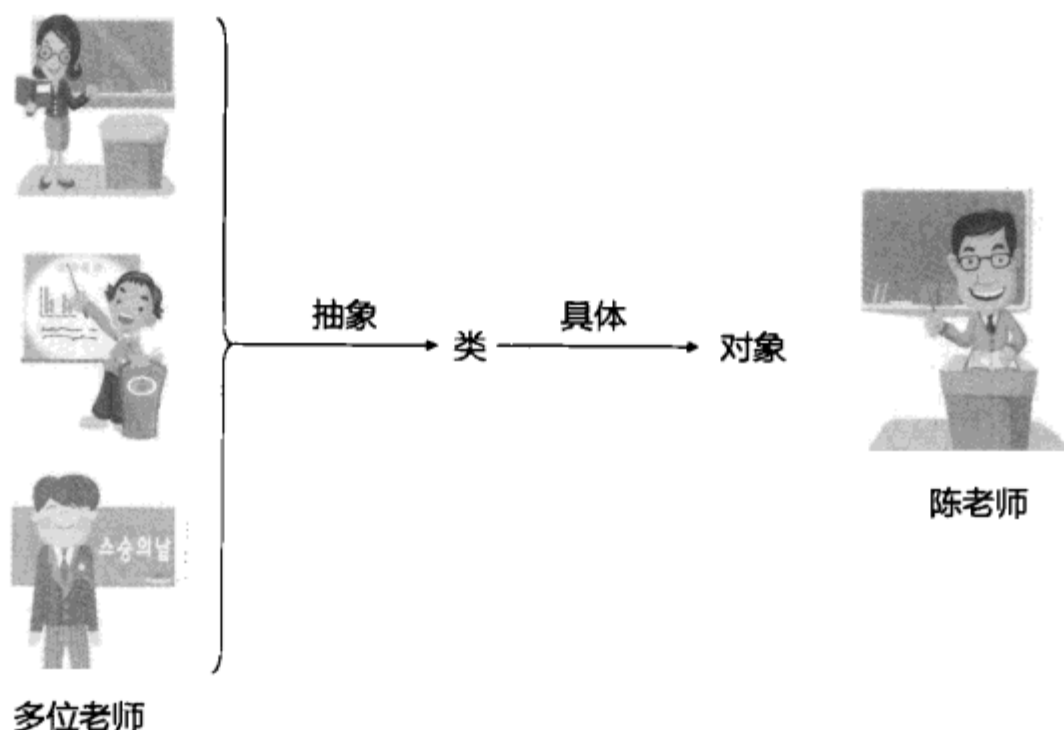


图 6-8 类与对象的关系

对象 vs. 物件

在中国台湾的一些面向对象思想的技术著作中，把 object 翻译成“物件”，那么 object-oriented 自然就成了“面向物件”。虽然把 object-oriented 翻译成“面向对象”已经约定俗成，但是“物件”一词更加准确地反映了 object 的实质：现实世界中的实际物体。

6.2.3 构造函数和析构函数

在现实世界中，每个事物都有生命周期，一个事物既然被创建，自然也会消亡。在 C++ 的世界中也这样，每个对象都有其生命周期：自创建时开始，到销毁时停止。一个对象的创

建和销毁，往往是其一生中非常重要的时刻，需要处理很多事情。例如，在创建对象的时候，需要进行很多初始化工作，设置某些属性的初始值；而在销毁的时候，可能需要进行一些清理工作，最重要的是把申请的资源释放掉，就像一个人离开人世时，应该把该还的钱还了，干干净净地走。为了完成对象的生与死这两件大事，C++中的类专门提供了两个特殊的函数——构造函数和析构函数，它们分别用来处理对象的创建和销毁。

由于构造函数在对象创建的时候被自动调用，所以这时我们可以完成很多不便在对象创建后进行的事情，比如可以在构造函数中利用一些特定的属性值来构造对象，完成对象的初始化工作，使得对象一旦被创建就有比较合理的初始值。就像人一出生就有了明确的性别一样，某些类的对象一旦被创建就需要有合适的初始值，这些初始值可以在构造函数中进行设定。所以，C++规定每个类都必须有构造函数，如果类没有声明构造函数，那么编译器会为其产生一个默认的构造函数，这个构造函数没有参数，也不做任何额外的事情。如果想在构造函数中完成一些特殊的任务，就需要自己为类添加构造函数。可以通过如下的方式为类添加构造函数：

```
class 类名
{
public:
    类名(参数列表)
    {
        // 对类进行构造，完成初始化工作
    }
}
```

因为构造函数具有特殊性，所以它的声明也比较特殊。首先构造函数没有返回值，而实际上构造函数返回的应该就是对象本身。其次，构造函数跟类同名，也就是用类的名字作为构造函数的名字。在构造函数中，我们可以拥有参数列表，利用这些参数可以对某个具体的类进行特殊的初始化，完成有差别的对象的定义。例如，上文中提到的 Teacher 类就没有显式地声明构造函数而是由系统为其产生一个默认的构造函数。下面改写这个 Teacher 类，添加一个构造函数来完成一些初始化的工作。

```
class Teacher
{
public:
    // 构造函数
    // 参数表示 Teacher 类对象的名字
    Teacher(string strName)    // 带参数的构造函数
    {
        // 使用参数对属性赋值，进行初始化
        // 使得这个对象一经创建，就有合适的姓名属性
        m_strName = strName;
    };
    void PrepareLesson();    // 备课
protected:
```

```

        string m_strName;           // 姓名
private:
};

```

现在就可以在定义对象的时候，通过带参数的构造函数给定这个对象的名字属性。

```

// 使用参数声明一个对象
Teacher MrChen("ChenLiangqiao");

```

在构造函数中，利用传递进来的参数对对象的姓名属性 `m_strName` 进行初始化，这样在定义 `Teacher` 对象时，就需要给定其名字参数，其得到的对象名字属性也是经过初始化后的。在上面的例子中，使用字符串“ChenLiangqiao”作为参数定义 `Teacher` 类的对象，定义完成后，`MrChen` 对象的名字属性的值就成为“ChenLiangqiao”。通过构造函数，可以给定对象中某些属性的初始值。

在构造函数中，除了可以使用赋值操作符“=”对对象的属性进行初始化之外，还可以使用初始化列表，直接利用构造函数的参数或者其他的合理初始值对对象的属性进行初始化。使用初始化列表对对象属性进行初始化的语法格式如下：

```

class 类名
{
public:
    // 使用初始化列表的构造函数
    类名(参数列表)
        : 属性1(初始值), 属性2(初始值) // 初始化列表
    {
    }
    // 类的其他声明和定义
};

```

其中，在构造函数后用“:”引出的就是初始化列表，它在对象创建的时候，也就是构造函数执行的时候，将使用属性后括号内的初始值完成属性的创建并对其进行初始化，这些初始值可以是构造函数的参数，也可以是属性的某个合理初始值。如果一个类有多个属性需要进行初始化，那么多个属性之间使用逗号分隔。例如，可以利用初始化列表将 `Teacher` 类的构造函数改写为：

```

class Teacher
{
public:
    // 使用初始化列表的构造函数
    Teacher(string strName)
        // 初始化列表，使用构造函数的参数 strName 创建并初始化属性 m_strName
        : m_strName(strName)
    {
    }

protected:

```

```
    string m_strName;
};
```

使用初始化列表改写后的构造函数，利用参数 strName 直接创建 Teacher 类的成员属性 m_strName 并对其进行初始化，这样就避免了使用赋值操作符“=”对 m_strName 进行初始化时所产生的多个临时对象的创建与销毁，可以在一定程度上提高对象构造的性能。所以，在可以的情况下，最好在构造函数的初始化列表中初始化类的所有成员属性。

这里需要注意的是，如果类已经有了显式定义的构造函数，编译器就不会再为其生成默认构造函数。例如，在 Teacher 类拥有显式声明的构造函数之后，如果还是采用如下的形式定义对象，就产生一个编译错误。

```
// 定义一个没有名字的老师
Teacher MrUnknown;
```

这时编译器就会提示错误，因为这个类已经没有默认的构造函数了，而唯一的构造函数需要给出参数。所以在设计类的时候，一般都会写出默认的构造函数，同时根据需要添加带参数的构造函数来完成一些特殊的初始化任务。

创建对象时需要构造函数，那么销毁对象时就需要析构函数了。析构函数是在对象的生命周期即将结束之前自动调用的，所以我们会在析构函数中处理一些对象被销毁前的清理工作。

析构函数跟构造函数相似，它的名称是由类名前加“~”构成的，同时析构函数也没有返回值。同样，如果类没有显式地声明析构函数，编译器也会自动为其产生一个析构函数。构造函数和析构函数的唯一不同在于析构函数不接受任何参数。下面来为 Teacher 类加上析构函数。

```
class Teacher
{
public:
    // ...
    // 析构函数
    ~Teacher()
    {
        // 进行清理工作
        cout<<"春蚕到死丝方尽，蜡烛成灰泪始干"<<endl;
    };

    // ...
};
```

因为 Teacher 类不需要额外的清理工作，所以在这个析构函数中我们没有定义任何操作，只是输出一段信息表示 Teacher 类对象的结束。一般来说，会将那些需要在对象被销毁之前自动完成的事情放在析构函数中来处理。例如，对象创建时申请的内存资源，需要在析构函

数中进行合理地释放，就像一个有信誉的人在离开人世之前要把欠别人的钱还清一样。

6.2.4 拷贝构造函数

在 C++ 中，除了可以使用构造函数直接创建一个新的对象之外，有时还需要根据已经存在的某个对象创建一个新的对象作为这个对象的副本。例如：

```
// 调用构造函数创建一个新对象 MrChen
Teacher MrChen("ChenLiangqiao");
// 利用 MrChen 对象创建一个新的对象 MissJia 作为其副本
Teacher MissJia(MrChen);
```

在这里，首先创建了一个 Teacher 类的新对象 MrChen，然后利用这个新对象作为 Teacher 类的构造函数的参数创建一个 MrChen 对象的副本 MissJia。我们将这种可以接受某个对象作为参数并创建一个新对象作为其副本的构造函数称为拷贝构造函数。拷贝构造函数实际上是构造函数的表亲，在语法格式上，两者基本相似，只是拷贝构造函数的参数是这个类的对象的引用，而它所创建的也正是作为参数的某个已经存在的对象的一个副本。默认情况下，如果一个类没有显式地定义其拷贝构造函数，编译器会为其创建一个默认的拷贝构造函数，以内存拷贝的方式将旧有对象的内存空间拷贝到新对象的内存空间，以此来完成新对象的创建。上面代码中 MissJia 这个对象的创建就是通过这种方式完成的。

在大多数情况下，默认的拷贝构造函数已经能够满足我们的需要了，但是有的时候，特别是类当中有指针类型的属性的时候，以拷贝内存形式实现的默认拷贝构造函数只能复制指针属性的值，而不能复制指针属性所指向的内存，在这种情况下，就需要自己定义类的拷贝构造函数，完成指针属性等需要特殊处理的属性的拷贝工作。例如，有一个 Computer 类，它有一个指针类型的属性 m_pKeyboard 指向它所连接的键盘。

```
// 键盘类
struct Keyboard
{
    // 键盘的型号
    string m_strModel;
};

// 定义了拷贝构造函数的电脑类
class Computer
{
public:
    // 默认构造函数
    Computer()
        : m_pKeyboard(nullptr)
    {}
    // 拷贝构造函数，参数是 const 修饰的 Computer 类的引用
    Computer(const Computer& com)
```



```

        : m_strModel(com.m_strModel) // 对象类型的成员属性直接使用初始化列表
                                         // 完成拷贝
    {
        // 获得已有对象 com 的指针属性 m_pKeyboard 并赋值给 pOldKeyboard
        Keyboard* pOldKeyboard = com.GetKeyboard();
        // 以 pOldKeyboard 指向的 Keyboard 对象为蓝本,
        // 创建一个新的 Keyboard 类对象赋值给 m_Keyboard 属性
        if( nullptr != pOldKeyboard )
            m_pKeyboard = new Keyboard(*(pOldKeyboard));
        else
            m_pKeyboard = nullptr;
    }

    // 省略析构函数

    // 成员函数
    void SetKeyboard(Keyboard* pKeyboard)
    {
        m_pKeyboard = pKeyboard;
    }
    Keyboard* GetKeyboard() const
    {
        return m_pKeyboard;
    }
private:
    // 指针类型的成员属性
    Keyboard* m_pKeyboard;
    // 对象类型的成员属性
    string m_strModel;
};

```

在这段代码中，我们为 Computer 类创建了一个自定义的拷贝构造函数。在这个拷贝构造函数中，针对对象类型的成员属性，我们直接使用初始化列表就完成了属性的拷贝。而针对指针类型成员属性 m_pKeyboard 的拷贝，并不能直接采用内存拷贝的形式完成，那样只是拷贝了指针的值，而指针所指向的内容并没有得到拷贝。要完成指针类型成员属性的拷贝，首先应该获得已有对象的 m_pKeyboard 属性，也就是获得它所指向的 Keyboard 对象，然后以这个 Keyboard 对象为蓝本，利用 Keyboard 类的默认拷贝构造函数创建这个 Keyboard 对象的一个副本，最后将其地址赋值给 m_pKeyboard 属性。这样，这个拷贝构造函数不仅能够拷贝 Computer 类的对象类型成员属性 m_strModel，也能够正确地完成指针类型成员属性 m_pKeyboard 的拷贝，最终才能完成 Computer 类的拷贝。例如：

```

// 引入断言所在的头文件
#include <assert.h>
//...

// 创建一个 Computer 对象 oldcom
Computer oldcom;
// 创建 oldcom 的 Keyboard 对象并修改其属性

```



```

Keyboard keyboard;
keyboard.m_strModel = "Microsoft-101";
oldcom.SetKeyboard(&keyboard);
// 利用 Computer 类的拷贝构造函数创建新对象 newcom
// 新的 newcom 对象是 oldcom 对象的一个副本
Computer newcom(oldcom);

// 使用断言 assert() 判断两个 Computer 对象的 m_pKeyboard 属性不同,
// 也就是它们分别指向两个不同的 Keyboard 对象,
// 但是, 这两个 Keyboard 对象的属性却是相同的
assert( newcom.GetKeyboard() != oldcom.GetKeyboard() );
assert( newcom.GetKeyboard()->m_strModel
        != oldcom.GetKeyboard()->m_strModel );

```

除了使用拷贝构造函数创建对象的副本作为新的对象之外, 在创建一个新对象之后, 还常常将一个已有的对象直接赋值给这个新对象来完成新对象的初始化。例如:

```

// 创建一个新的对象
Computer newcom;
// 利用一个已有的对象对其进行赋值, 完成初始化
newcom = oldcom;

```

跟类的拷贝构造函数相似, 如果没有显式地为类定义赋值操作符, 编译器也会为其生成一个默认的赋值操作符, 以内存拷贝的形式完成对象的赋值操作。因为同样是以内存拷贝的形式完成对象的复制, 所以当类中有指针型属性时, 也会遇到跟拷贝构造函数相同的无法拷贝指针所指向的内存的问题。因此, 要完成带有指针型属性的类对象的赋值, 必须对赋值操作符进行自定义。例如, 可以自定义 Computer 类的赋值操作符来完成指针型属性的赋值。

```

// 定义赋值操作符 "=" 的电脑类
class Computer
{
public:
    // 自定义的赋值操作符
    Computer& operator = (const Computer& com)
    {
        // 判断是否是自己给自己赋值
        // 如果是自赋值, 则直接返回对象本身
        if( this == &com ) return *this;

        // 直接完成对象型属性的赋值
        m_strModel = com.m_strModel;
        // 使用 "new" 创建旧有对象的副本, 完成指针型属性的赋值
        m_pKeyboard = new Keyboard(*(com.GetKeyboard()));
    }
    // ...
};

```

在 Computer 类的自定义赋值操作符中, 我们首先判断这是否是一个自赋值操作。所谓自赋值, 就是自己给自己赋值。例如:

```
// 自赋值操作
newcom = newcom;
```

当赋值操作符检测到这种自赋值操作时，因为这种自赋值操作是没有意义的，所以它就直接返回这个对象本身。如果不是自赋值操作，对于对象型属性，则使用赋值的方式直接完成其赋值；而对于指针型属性，则采用跟拷贝构造函数相似的方式，通过创建旧有对象的副本来完成其赋值。

6.2.5 操作符重载

在 6.2.4 小节中，赋值操作符“=”的自定义实际上是操作符的重载，也就是把已经定义的、有一定功能的操作符进行重新定义，来完成更为细致具体的运算等功能。操作符重载可以将概括性的抽象操作符具体化，便于外部调用而无须知晓其内部的具体运算过程。

C++有许多内置的数据类型，包括 int、char、string 等，每种类型都有许多运算符，例如 +、-、*、/ 等，可以利用这些运算符方便地对两个数据进行运算，从而得到相应的运算结果。比如，可以使用乘法运算符“*”对两个 int 类型的变量进行运算得到两个数的积；也可以使用加法运算符“+”对两个 string 类型的变量进行运算，简单方便地得到两个字符串合并后的结果。例如：

```
int a = 3;
int b = 4;
// 使用乘法运算符“*”获得两个 int 类型变量的乘积
int c = a * b;
string strSub1("Hello");
string strSub2("C++");
// 使用加法运算符“+”获得两个 string 类型变量的合并结果
string strCombin = strSub1 + strSub2;
```

针对这些内置的数据类型，C++提供了丰富的运算符供我们完成常见的操作。但是当我们定义了新的类的对象时，两个对象之间是不能进行这些操作的。比如分别定义了 Father 类和 Mother 类的两个对象，希望这两个对象也可以使用加法运算符“+”运算得出一个 Baby 类的对象。

```
// 分别定义 Father 类和 Mother 类的对象
Father father;
Mother mother;
// 通过加法运算符“+”得到 Baby 类的对象
Baby baby = father + mother;
```

以上语句所表达的是一件显而易见的事情，但是，如果没有重载 Father 类的加法运算符“+”，编译器是不知道如何将 father 对象和 mother 对象加起来创建一个 baby 对象的，而这样的语句会出现编译错误。但幸运的是，C++允许我们对这些操作符进行重载，告诉编译器该如何进行自定义对象之间的各种运算。重载操作符之后，可以方便地在类对象之间使用操

作符进行各种有意义的运算，就像内置类型的运算一样方便。在功能上，重载操作符等同于类的成员函数，两者并无明显差别，可以简单地讲重载操作符是一类比较特殊的成员函数。虽然成员函数可以提供跟操作符相同的功能，但是运用操作符可以让语句更加简洁，也更加易懂。比如，“a.add(b)”调用函数 add()以实现两个对象 a 和 b 相加，但是实现相同功能的“a + b”语句，远比“a.add(b)”更容易让人理解。

在 C++ 中，声明重载操作符的语法格式如下：

```
class 类名
{
public:
    返回值类型 operator 操作符 (参数列表)
    {
        操作符的具体运算过程
    }
};
```

从这里可以看到，重载操作符和成员函数虽然在功能上是相同的，但是在语法格式上还是存在一些细微的差别。普通成员函数以标识符作为函数名，而重载操作符以“operator 操作符”作为函数名。在使用上，当使用操作符对两个对象进行运算时，实际上是调用第一个对象的操作符，而第二个对象则作为这个操作符的参数。例如，使用加法运算符对两个对象进行运算：

```
a + b;
```

这条语句实际上等同于：

```
a.operator + (b);
```

它表示调用的是对象 a 的操作符“operator +”，而参数则是对象 b。当调用类的重载操作符时，编译器会将调用该操作符的第一个对象作为操作符的调用者，也就是隐含的 this 指针。可以通过 this 指针访问调用该操作符的第一个对象。例如，可以这样定义 Father 类的加法运算符，以实现上面两个对象相加的语句：

```
// 母亲类
class Mother
{
    // 省略声明和定义
};
// 孩子类
class Baby
{
public:
    // 孩子类的构造函数
    Baby(string strName)
        : m_strName(strName)
    {}
private:
```

```

        // 孩子的名字
        string m_strName;
    };
    // 父亲类
    class Father
    {
    public:
        // 重载操作符 "+"
        Baby operator + (const Mother& mom)
        {
            // 创建一个 Baby 对象并返回
            return Baby("ChenXibei");
        }
    };

```

在 Father 类的重载操作符“+”中，它可以接受一个 Mother 类的对象作为参数，并在其中创建一个 Baby 类的对象作为操作符的返回值。这样就完整地表达了一个 Father 类的对象加上一个 Mother 类的对象得到一个 Baby 类的对象的意义，现在就可以方便地使用操作符“+”将 Father 类的对象和 Mother 类的对象相加而得到一个 Baby 类的对象了。

6.2.6 类成员的访问控制

类成员包括类的成员变量和成员函数，它们分别用来描述类的属性和行为。而类成员的访问控制决定哪些成员是公开的，可以被外界访问；哪些成员是私有的，只能在类的内部访问，外界无法访问。就像一个人盘子中的奶酪，只能自己动，别人是不能动的。

大家可能会问，为什么类这么自私自利，还要对类成员的访问加以控制，大公无私不是挺好的吗？可以设想这样的情形：名字是一个人的属性，当然也会是你的属性。一般而言，只有你老爸和你自己有权修改你的名字。如果其他人随便修改你的名字，恐怕你不乐意。所以要对对象的属性和行为的访问加以控制，以此来保护某些属性和行为不被非法访问。

在 C++ 中，对类成员的访问控制是通过设置成员的访问级别来实现的。访问级别按照访问的范围不同分为公有类型（public）、保护类型（protected）和私有类型（private）三种，如图 6-9 所示。

1. 公有类型

公有类型的成员用关键字 public 表示，在成员变量或者成员函数前加上 public 就表示这是一个公有类型的成员。公有类型的成员是可以供外界访问的，它是类提供给外界访问的接口，是类跟外界交流的通道，外界通过访问公有类型的成员完成跟类的交互。同时在类的内部也可以访问到公有类型的成员。



图 6-9 访问级别

2. 保护类型

保护类型的成员用关键字 `protected` 表示, 其声明格式跟 `public` 类型的相同。保护类型的成员可以在类的内部及类的派生类中访问, 它主要用于将属性或者方法遗传给它的下一代子类。类的外部无法访问保护类型的成员。

3. 私有类型

私有类型的成员用关键字 `private` 表示, 其声明格式跟 `public` 类型的相同。私有类型的成员只能在类的内部被访问, 所有来自外部的访问都是非法的, 这样就把类的成员完全隐藏在类当中, 很好地保护了类中数据和行为的安全。所以, 赶快把我们的钱包声明为私有成员吧, 这样小偷就不会访问到它了。

这里需要说明的是, 如果成员没有显式地说明它的访问级别, 那么默认情况下它就是私有类型, 外界是无法访问的, 由此可见类是非常“自私”的。

我们把以上例子中的 `Teacher` 类根据访问控制进行改写, 以更加真实地反映现实的情况。

```
// 添加访问控制后的 Teacher 类
class Teacher
{
// 公有类型成员
// 外界通过访问这些成员跟该类进行交互
public:
// 构造函数应该是公有的, 这样外界才可以利用构造函数创建该类的对象,
// 否则, 只有自己创建自己了
Teacher(string strName)
{
    m_strName = strName;
}
// 老师要为学生们备课, 它会被外界调用, 所以这个行为是公有类型的
void PrepareLesson()
```

```

{
    cout<<"老师备课。"<<endl;
}
// 我们不让别人修改名字，可总得让别人知道我们的名字吧，
// 所以提供一个成员函数供外界获得我们的名字
string GetName()
{
    return m_strName;
}
// 保护类型成员
// 不能被外界访问，但是可以遗传给下一级子类，
// 供下一级子类访问
protected:
    // 自己的名字，好好保护起来
    string m_strName;
private:    // 私有类型
};

```

现在再进行 Teacher 类访问的时候，就要注意类成员的访问控制了，稍不留神就会被拒之门外，吃人家的闭门羹。

```

int _tmain(int argc, _TCHAR* argv[])
{
    // 创建对象时会调用类的构造函数
    // 构造函数是公有类型，所以可以直接调用
    Teacher MrChen("ChenLiangqiao");

    // 外部变量，用于保存从对象获得的数据
    string strTeacherName;
    // 直接调用类的公有类型成员函数，获得类中的数据
    strTeacherName = MrChen.GetName();

    // 错误：无法直接访问类的保护类型成员
    // 想改我的名字？先要问我答应不答应
    MrChen.m_strName = "Jiawei";

    return 0;
}

```

在主函数中，我们首先创建一个名字属性为“ChenLiangqiao”的对象，虽然该属性是保护类型的，但是可以在构造函数中对该属性进行修改。其次，为了让外界能够安全地读取该属性的值，我们为 Teacher 类添加了一个公有类型的成员函数 GetName()，可以利用该成员函数来获得 Teacher 对象的保护类型的属性 m_strName。但是，如果想通过直接访问该保护类型的成员变量去修改该对象的名字属性，编译器会帮我们检测出这个非法访问，产生一个编译错误提示无法访问保护类型的成员。这样，有编译器帮我们看着，别人就没法动我们的奶酪了。

通过对类成员的访问进行控制，很好地起到了保护数据的作用，防止数据被外界随意修改。如果对象的某些属性因为业务逻辑的需要允许外界访问，也建议采用提供公有接口的方法。

法，让外界通过公有接口来访问这些属性，而不是直接把这些属性声明为公有类型。一般情况下，类的属性都应该声明为保护类型或者私有类型。

6.2.7 在友元中访问类的隐藏信息

采用类成员的访问控制机制之后，很好地实现了数据的隐藏与封装，类的成员变量一般定义为私有成员，成员函数一般定义为公有成员，以此来提供类与外界间的通信接口。但是，严格的成员访问控制有时也会带来一些麻烦。例如，有时需要定义某个函数，这个函数不是类的一部分，但又需要频繁地访问类的隐藏信息（包括受保护的和私有的成员变量和成员函数）；又或者需要定义某个新的类，因为某种原因，这个类需要访问另外一个类的隐藏信息。在这些情况下，我们需要从外界直接访问类的成员变量，但是严格的成员访问控制使得这成为几乎不可能完成的任务。

但是，凡事都有例外。为了让外界的函数或者类能够访问某个类的隐藏信息，C++提供了一个“friend”关键字来完成这些不可能完成的任务。利用“friend”关键字，可以将外界的一个函数或者类声明为类的友元函数或者友元类，两者统称为友元。

1. 友元函数

友元函数实际上是一个定义在类外部的普通函数，它不属于任何类。当使用“friend”关键字在类的定义中加以声明时，这个函数就成为类的友元函数。成为友元函数之后，它就可以不受类成员访问控制的限制，直接访问类的隐藏信息。在类中声明友元函数的语法格式如下：

```
class 类名
{
    friend 返回值类型 函数名(形式参数);
    // 类的其他声明和定义...
};
```

友元函数的声明跟类的成员函数的声明是相同的，只是友元函数定义在类的外部，不属于类的成员函数。友元函数的声明既可以放在类的私有部分，也可以放在类的公有部分，它们是没有区别的，都说明是该类的一个友元函数。同时，函数可以是多个类的友元函数，只是需要在各个类中分别声明。

2. 友元类

跟友元函数相似，友元类是定义在另一个类之外的普通类。因为需要访问另一个类的隐藏信息，所以利用“friend”关键字将其声明为另一个类的友元类，赋予它访问另一个类的隐藏信息的能力。成为另一个类的友元类之后，友元类的所有成员函数都成为另一个类的友元函数，都可以访问另一个类中的隐藏信息。

在 C++ 中，声明友元类的语法格式如下：

```
class 类名
{
    friend class 友元类名;
    // 类的其他声明和定义
};
```

经过这样的声明后，在友元类中就可以不受类成员访问控制的限制，直接访问另一类的隐藏信息。为了更好地理解友元的作用，还是来看一个实际的例子。假设在定义的 Teacher 类中有一个属性 m_nSalary 记录了老师的工资信息。工资信息当然是个人隐私，应该将其声明为受保护的成员属性。

```
class Teacher
{
    // 工资信息：受保护的成员属性
protected:
    int m_nSalary;
};
```

但是，在某些特殊情况下，我们又不得不访问 Teacher 类中受保护的工资信息属性 m_nSalary。比如，税务局 (TaxationDep) 要来查老师的工资收入，它当然有权力也有必要访问 Teacher 类中 m_nSalary 这个受保护的成员属性；又或者学校想用 AdjustSalary() 函数给老师调整工资，老师当然乐意它来访问 m_nSalary 这个受保护的成员属性。在这种情况下，我们只好利用友元来帮忙了。

```
// 拥有友元的 Teacher 类
class Teacher
{
    // 声明 TaxationDep 类为 Teacher 类的友元类
    friend class TaxationDep;
    // 声明 AdjustSalary() 函数为 Teacher 类的友元函数
    friend int AdjustSalary(Teacher& teacher);
protected:
    int m_nSalary;
};

// 友元函数
int AdjustSalary(Teacher& teacher)
{
    // 在 Teacher 类的友元函数中访问 Teacher 类的受保护的成员
    return teacher.m_nSalary + 299;
}

// 友元类
class TaxationDep
{
public:
    int CheckSalary( Teacher& teacher )
    {
```

```
        // 在 Teacher 类的友元类中访问 Teacher 类的受保护成员
        return teacher.m_nSalary;
    }
};
```

可以看到，当 Teacher 类利用“friend”关键字将 AdjustSalary()函数和 TaxationDep 类声明为它的友元之后，就可以在友元中直接访问它的受保护的成员。必要的时候，使用友元可以帮助我们解决很多无法直接访问类的隐藏信息的问题。

友元虽然能给我们带来便利，但是在使用友元类的时候，还应该注意以下几点：

- 友元关系不能被继承。这一点很好理解，我们跟某个类是朋友（即是某个类的友元类），并不表示我们跟这个类的儿子（派生类）同样是朋友。
- 友元关系是单向的，不具有交换性。比如，TaxationDep 类是 Teacher 类的友元类，税务官员可以检查老师的工资，但是这并不表示 Teacher 类也是 TaxationDep 类的友元类，老师也可以检查税务官员的工资。

友元的使用并没有破坏封装

在友元函数或者友元类中，我们可以直接访问类的受保护的（protected）或者私有的（private）成员，很多人担心这样会让类的隐藏信息暴露出来，破坏类的封装。但事实上，合理地使用友元，不仅不会破坏封装，反而会增强封装。

在面向对象的设计中，我们强调的是“高内聚、低耦合”的设计原则。当一个类的两部分有不同数量的实例或者不同的生命周期时，为了保持类的“高内聚”，经常需要将一个类分割成两部分，也就是将一个类分割成两个类。在这种情况下，两部分通常需要直接存取彼此的数据。实现这种情况的最安全途径就是使这两个类成为彼此的友元。但是，一些“高手”想当然地认为友元破坏了类的封装，转而通过提供公有的 get()和 set()成员函数使两部分可以访问数据。实际上，他们不知道这样的做法正是破坏了封装。在大多数情况下，这些 get()和 set()成员函数和公有数据一样差劲：它们仅仅隐藏了私有数据的名称，而没有隐藏私有数据本身。

友元的使用，它只是向必要的客户公开类的隐藏数据，比如 Teacher 类只是向 TaxationDep 类公开它的隐藏数据，这比使用公有的 get()/set()成员函数来访问 Teacher 类的数据有更好的封装性。

6.3 类如何面向对象

类作为 C++与面向对象思想结合的产物，它的身上流淌着面向对象的血液。从类成员的构成到类之间的关系，到处都体现着面向对象封装、继承和多态的思想。

6.3.1 用类机制实现封装

类是对现实世界中事物的描述。学校中每个老师的名字、年龄等属性都不相同，同时每个老师的特长又各不相同，有的擅长唱歌，有的擅长弹琴。假如要用类来描述学校的所有老师，那么我们是不是要对每个老师都建立一个类呢？

答案当然是否定的。面向对象程序设计思想可以采用抽象的方法，对现实世界中的多个具体对象（多个老师）进行概括分析，得到这类对象所具有的公共属性和行为，加以描述就形成了类。虽然都是同一个类的对象（老师），但是每个对象的属性不同，就形成了不同的具体对象实体（各个老师）。

抽象一般分为属性抽象和行为抽象两种。前者寻找一类对象共有的属性或者状态变量，比如老师的年龄、姓名等，然后将其作为类的成员变量；而后者则寻找这类对象所具有的共同行为特征，比如老师都会备课、上课等，最终成为类的成员函数。当以后我们分析新的对象时，都会从属性和行为两个方面进行抽象和概括，提取对象的共有特征。而整个抽象过程，是一个从具体到一般的过程。

如果说抽象是将很多对象的共有特征提取出来成为类的成员属性和成员函数，那么封装机制则是将这些特征进行有机地结合形成一个完整的类。在 C++ 语言中，我们可以使用 6.2 节中介绍的类（class）概念来封装某类对象的属性和行为。比如老师这类对象就可以封装为：

```
// 用 Teacher 类封装老师的属性和行为
class Teacher
{
// 构造函数
public:
    Teacher(string strName)
    {
        m_strName = strName;
    };
// 用成员函数描述老师的行为
public:
    void PrepareLesson();    // 备课
    void TeachLesson();     // 上课
    void ReviewHomework();  // 批改作业

// 用成员变量描述老师的属性
protected:
    string    m_strName;    // 姓名
    int       m_nAge;       // 年龄
    bool      m_bMale;      // 性别
    int       m_nDuty;      // 职务
private:
};
```

通过封装，可以将老师这类对象的属性和行为紧密结合在 Teacher 类中，形成一个可重用的程序模块。现在就可以用 Teacher 类的某个对象来描述某位具体的老师。例如：

```
// 学校中的某位陈老师  
Teacher MrChen("ChenLiangqiao");  
// 学校中的某位贾老师  
Teacher MissJia("Jiawei");
```

虽然 MrChen 和 MissJia 这两个对象都属于 Teacher 类的对象，但是因为它们的属性不同，所以可以描述现实世界中的两位不同的老师。

封装好的类通过提供特定的外部接口，可让其他对象调用自己的行为完成任务，但是对外隐藏了对象行为的实现细节。另外，在大多数情况下，外界是无法直接访问对象的属性的，这样就很好地实现了对数据的隐藏，如图 6-10 所示。

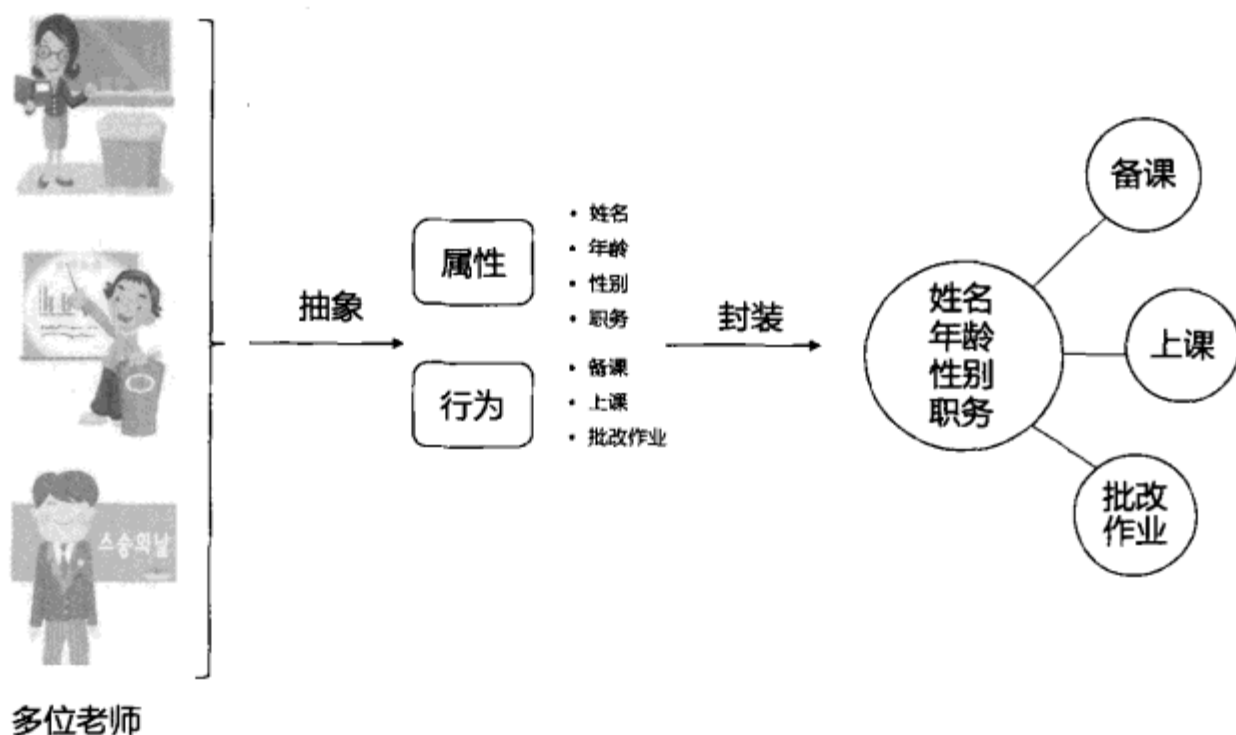


图 6-10 抽象与封装

抽象与封装，用来将现实世界的事物转变成 C++ 世界中的各个类，也就是用程序语言来描述现实世界。结构化程序设计中也有抽象这个过程，只是它的抽象仅针对现实世界中的数据，而面向对象的抽象不仅包括事物的数据，还包括事物的行为，更进一步地，面向对象利用封装将数据和行为有机地结合在一起，从而更加真实地反映现实世界。抽象与封装，完成了从现实世界中的具体事物到 C++ 世界中一般类的过程，是将现实世界程序化的有力武器。

6.3.2 用基类和派生类实现继承

在理解了面向对象思想的抽象与封装后，继续分析上面的例子。在现实世界中，我们发现老师和学生这两类不同的对象有一些相同的属性和行为，比如都有姓名、年龄、性别，都

可以走路、说话、吃饭等。为什么不同的对象可能会有相同的属性和行为呢？这是因为这些特征都是人类所共有的属性和行为，它们都是人类，所以都具有这些共同的属性和行为。在现实世界中也有很多相类似的情况，比如小汽车、卡车是汽车的某个子类别，它们都具有汽车的属性和行为；三角形、正方形都是图形的具体化，它们都具有图形的属性和行为。为了描述现实世界中的这种关系，C++提供了继承的机制，允许我们在保持原有对象特性的基础上进行更加具体的说明或者扩展，从而形成新的类别。例如，可以说“会上课的人就是老师”，实际上就是在人这个基础上添加一个“会上课”的行为就形成了“老师”这个新的类。老师这个类从人类中继承了大量的属性和行为，所以老师这个类同样有姓名、年龄等人类拥有的属性以及走路、吃饭等人类拥有的行为，同时它还在这个基础上添加了自己特有的行为：上课。这样既有继承也有发展，构成了一个新的老师类。

所谓继承，就是从父辈处得到的属性和行为。同时，继承又不仅仅是完全照搬父辈的属性和行为，而是通过继承对父辈的属性和行为进行进一步的细化或者扩充来形成新的类。这样，当复用旧有的类形成新类时，只需要从旧有的类继承，然后修改或者扩充需要的属性和行为即可。新类复用旧有对象的属性和行为，体现了面向对象的共享机制。在C++中，我们把旧有的类称为基类或者父类，而把从基类继承产生的新类则称为派生类，有时也可以形象地称为子类。下面来看一个实际的例子，如图6-11所示。

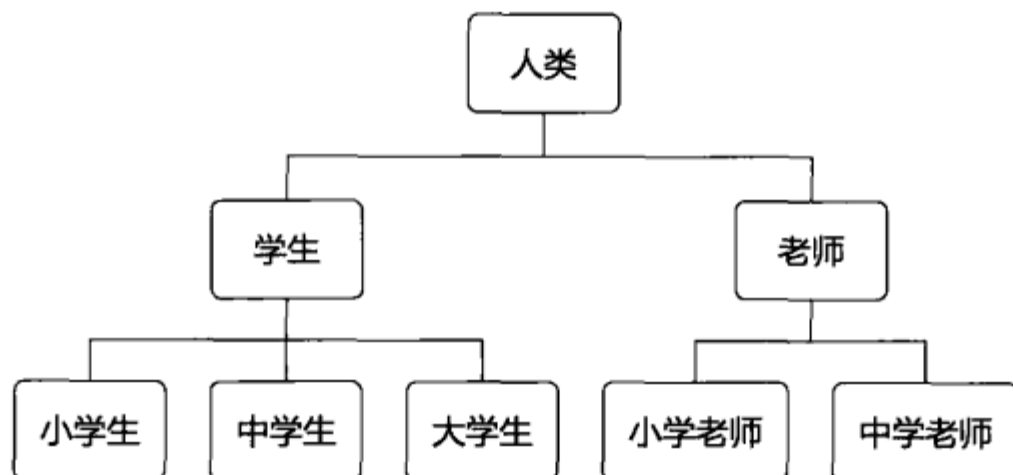


图 6-11 现实世界的继承关系

从这棵继承树中可以看到，老师和学生都继承自人类，这样，老师和学生就具有了人类的属性和行为，而小学生、中学生、大学生继承自学生这个类，他们不但具有人的属性和行为，同时还具有学生的属性和行为。通过继承，派生类不用再去重复设计和实现基类已有的属性和行为，只要直接通过继承拥有基类的属性和行为，从而实现设计和代码最大限度上的复用。

在C++中，派生类的声明方式如下：

```
class 派生类名 : 继承方式 基类名 1, 继承方式 基类名 2...  
{  
    // 派生类新增加的属性;
```

```
    // 派生类新增加的行为;  
};
```

其中，基类名是已经声明的类的名字。如果一个类只有一个基类，这种情况称为单继承。如果一个派生类可以有多个基类，这种情况称为多继承，这时派生类可以同时得到多个基类的特征，就如同我们身上既有父亲的特征，也有母亲的特征一样。

跟类成员的访问控制类似，派生方式也有 public、protected 和 private 三种。不同的派生方式决定了派生类如何访问从基类继承下来的成员变量和成员函数。

1. 派生方式

(1) public。

public 派生被称为类型继承，表示派生类是基类的一个子类型，而基类所有成员的访问级别在派生类中不做改变。public 派生反映了派生类和基类之间的一种“is-a”的关系，例如 Teacher 类从 Human 类派生，那么可以说老师 (Teacher) 是一个人 (Human)。大多数情况下，我们都采用 public 方式的继承。

(2) private。

private 派生被称为实现继承，它把基类的所有公有 (public) 成员都变成自己的私有 (private) 成员，这样，派生类将不再直接支持基类的公有接口，它只希望可以重用基类的实现而已。

(3) protected。

protected 派生把基类的所有公有成员变成 protected 类型，以此来保护基类的所有接口不被外界访问，只能由自身及自身的派生类访问。

在了解了派生类的声明方式后，就可以用具体的代码来描述上面这棵继承树所表达的继承关系了。

```
// 定义基类 Human  
class Human  
{  
    // 人类共有的行为  
public:  
    void Walk();           // 走路  
    void Talk();           // 说话  
    // 人类共有的属性  
protected:  
    string  m_strName;      // 姓名  
    int     m_nAge;         // 年龄  
    bool    m_bMale;        // 性别  
private:
```

```

};

// public 继承方式继承 Human 类
// 表示 Teacher 跟 Human 是 “is-a” 的关系
class Teacher : public Human
{
// 在子类中添加老师特有的行为
public:
    void PrepareLesson();        // 备课
    void TeachLesson();          // 上课
    void ReviewHomework();       // 批改作业
// 在子类中添加老师特有的属性
protected:
    int    m_nDuty;              // 职务
private:
};

// public 继承方式继承 Human 类
class Student : public Human
{
// 在子类中添加学生特有的行为
public:
    void AttendClass();          // 上课
    void DoHomework();           // 做家庭作业
// 在子类中添加学生特有的属性
protected:
    int m_nScore;
private:
};

// public 继承方式继承 Student 类
class Pupil : public Student
{
// 在子类中添加小学生特有的行为
public:
    void AttendClass();          // 上课
    void DoHomework();           // 做家庭作业
protected:
private:
};

```

在这段代码中，首先声明了人（Human）这个类，它定义了人这类对象应当具有的共有属性（姓名、年龄、性别）和行为（走路、说话）。因为老师这个类是人的一种，是人整个类的具体化，所以我们以 Human 这个类为基类，以 public 继承的方式定义 Teacher 这个派生类。通过继承，Teacher 这个类不仅直接具有了 Human 类的所有属性和行为，同时还根据需要添加了 Teacher 类自己所特有的属性（职务）和行为（备课、上课），这样就完成了对 Human 类的继承和扩展，从而形成了新的 Teacher 类。现在通过继承，Teacher 类已经是一个“会教书的人类”了。


```
// 定义一个 Teacher 对象
Teacher MrChen;
// 老师走进教室
// 我们在 Teacher 类中并没有定义 Walk() 成员函数,
// 这里是通过继承从基类 Human 中得到的成员函数
MrChen.Walk();
// 老师开始上课
// 这里调用的是 Teacher 自己定义的成员函数
MrChen.TeachLesson();
```

同理,我们还定义了 Student 类。定义好 Student 类后,又根据需要定义了更加具体的 Pupil 类。

仔细体会就会发现,整个继承的过程就是类的不断具体化、不断传承基类的属性和行为,并且发展自己特有属性和行为的过程。而这个基类和派生类之间继承的过程,体现在现实世界中,就是事物不断发展进化的过程。

生物物种之间的进化,是依靠吸收和保留部分父辈的能力,同时根据环境的变化,对父辈的能力做一些改进并增加一些新的能力来完成的。面向对象思想中的继承是进化过程在程序世界中的反映,它同样依靠这三种方式来完成从基类到派生类的进化。

2. 进化方式

(1) 保留基类的属性和行为。

继承最大的目的就是复用基类的设计和代码,保留基类的属性和行为,而不用自己白手起家,一切从零开始。在上面的例子中,派生类 Teacher 通过继承 Human 类,轻松拥有了 Human 类的所有属性和行为,除了 Human 类的构造函数和析构函数之外,其他类成员都全部保留在了 Teacher 类中,这就像站在巨人的肩膀上,Teacher 类只用很少的代码就拥有了基类的所有成员,实现了设计和代码的复用。

(2) 改进基类的属性和行为。

既然是进化,派生类就要有优于基类的地方,这些地方就表现在派生类对基类属性和行为的修改。例如,Student 类和 Pupil 类都有 DoHomework() 这个成员函数,但是这两个类对这个函数的实现是不同的,派生类 Pupil 会根据自己的实际情况对这个函数做进一步的具体化,对这个行为进行改写以适应新的需求。

(3) 添加新的属性和行为。

如果进化仅仅是对原有事物的改进,那么是远远不够的。进化还需要一些“革命性”的东西才能产生新的事物。所以在类的继承当中,仅仅改进基类的属性和行为是不够的,还需要添加一些“革命性”的新的属性和行为。例如,Teacher 类从 Human 类派生,它不仅保留

了基类的属性和行为，同时还根据需要添加了基类所没有的新属性（职务）和行为（备课、上课），正是这些新添加的属性和行为，使它从本质上区别于 Human 类，完成了从 Human 到 Teacher 的进化。

很显然，继承既很好地解决了设计和代码复用的问题，又提供了一种扩展的方式来轻松应对新的需求，而这正是面向对象思想的魅力所在。

既然继承可以带来这么多好处，不用费很大的劲就可以复用以前的设计和代码，那么是不是可以在能够使用继承的地方就都使用继承，而且越多越好呢？

当然不是。人参再好，也不能当饭吃。正是因为继承太有用，带来了很多好处，所以往往会被初学者滥用，最后导致设计出一些“四不像”的怪物出来。在这里，我们要给继承的使用定几条规矩。

3. 继承使用的规范

(1) 拥有派生关系的两个类必须相关。

如果两个类（A 和 B）毫不相关，则不可以为了使 B 的功能更多而让 B 继承 A 的功能。也就是说，不可以为了让“人”具有“飞行”的行为，而让“人”从“鸟”派生，那就不再是“人”，而是“鸟人”了。不要觉得类的功能越多越好，在这里，要奉行“多一事不如少一事”的哲理。

(2) 不要把组合当成继承。

如果类 B 有必要使用类 A 的功能，则要分两种情况考虑。

- 1) B 是 A 的“一种”。若在逻辑上 B 是 A 的“一种”（a kind of），则允许 B 继承 A 的功能。例如，老师（Teacher）是人（Human）的一种，那么 Teacher 就可以从 Human 继承。
- 2) A 是 B 的“一部分”。若在逻辑上 A 是 B 的“一部分”（a part of），则不允许 B 继承 A 的功能，而是要用 A 和其他东西组合成 B。例如键盘（keyboard）、鼠标（mouse）和显示器（monitor）都是电脑（computer）的组成部分，虽然可以说一台电脑拥有键盘、鼠标、显示器的功能，但是这里的电脑不能由键盘、鼠标、显示器派生，而应该由它们组合而成。

```
// 键盘
class Keyboard
{
public:
    // 接收用户键盘输入
    void Input();
};
```

```

// 鼠标
class Mouse
{
public:
    // 鼠标单击
    void Click();
};
// 显示器
class Monitor
{
public:
    // 显示画面
    void Display();
};
// 电脑
class Computer
{
// 电脑的行为
// 电脑的具体行为都由其各个组成部分来负责完成
public:
    // 用键盘、鼠标和显示器组合一台电脑
    Computer( Keyboard* pKeyboard,
              Mouse* pMouse,
              Monitor* pMonitor )
    {
        m_pKeyboard = pKeyboard;
        m_pMouse = pMouse;
        m_pMonitor = pMonitor;
    }
    // 键盘负责用户输入
    void Input()
    {
        m_pKeyboard->Input();
    }
    // 鼠标负责用户单击
    void Click()
    {
        m_pMouse->Click();
    }
    // 显示器负责显示画面
    void Display()
    {
        m_pMonitor->Display();
    }
// 电脑的各个部件
private:
    Keyboard* m_pKeyboard;    // 键盘
    Mouse* m_pMouse;         // 鼠标
    Monitor* m_pMonitor;     // 显示器
};

```

在上面的代码中，电脑这个类由三个部件组成。它的所有功能都不是自己实现的，而是

由它的各个部件实现的，它只是提供了一个统一的对外接口而已。这种把几个类综合在一起构成新类的方式就是组合。为了让电脑具有各个部件的功能而让它从各个部件继承，那它会是一个什么样的电脑呢？即使能用恐怕也是一个怪物。

关于组合，还需要注意的是，这里使用了对象指针来把各个部分联系起来，是因为电脑是一个可以插拔的系统。键盘、鼠标、显示器都是可以更换的，键盘可以在这台电脑上使用，也可以在另外的电脑上使用，电脑和键盘的生命周期是不同的。所以这里采用指针，两个对象可以各自独立地创建后再组合起来。如果遇到整体和部分密不可分的情况，两者具有相同的生命周期，比如一个人和组成这个人的胳膊、大腿等，这时就该采用对象了。

6.3.3 用虚函数实现多态

在理解了面向对象的继承机制之后，我们知道了在大多数情况下派生类是基类的“一种”，就像学生是人中的一种一样。换句话说，学生是人的一种，那么在使用“人”的时候，这个“人”可以是“学生”，而“学生”也可以应用在“人”的场合。比如可以问“教室里有多少人”，实际上问的是教室里有多少学生。这种用基类指代派生类的关系反映到 C++ 中，就是基类指针可以指代派生类的对象，而派生类的对象也可以当成基类对象使用。这样的解释对大家来说是不是很抽象呢？没关系，可以回想生活中经常遇到的场景：“上车的人请买票”。在这句话中，涉及一个类——人，以及它的一个动作——买票。上车的人可能是老师、学生，也可能是工人、农民或者某个程序员，为什么售票员不说“上车的老师请买票”或者说“上车的工人请买票”，而仅仅说“上车的人请买票”就足够了？这是因为“人”是基类，虽然上车的人可能是老师、学生、公司职员等，但是他们都是“人”这个基类的派生类，所以这里就可以用基类“人”来指代所有派生类对象，如图 6-12 所示。

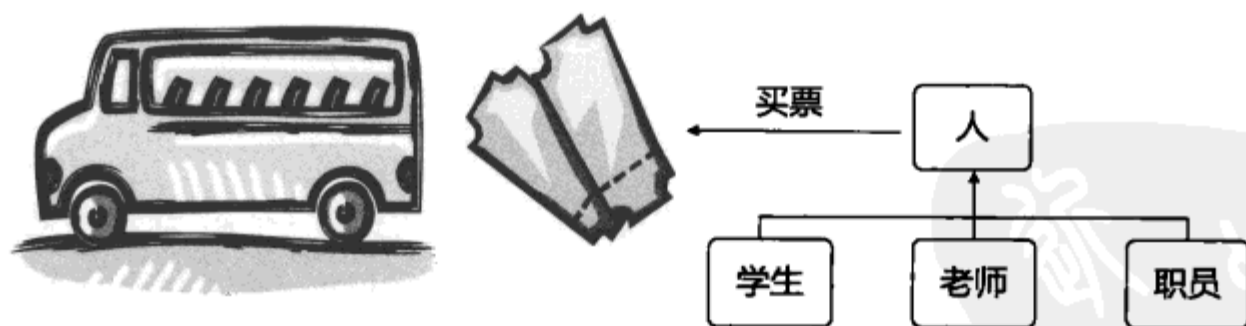


图 6-12 “上车的人请买票”

下面用实际的程序来描述这个场景。

```
// “上车买票”演示程序
// 定义 Human 类，这个类有一个成员函数 BuyTicket() 表示买票的动作
class Human
{
// Human 类的行为
```

```

public:
    // 买票
    void BuyTicket()
    {
        cout<<"人买票。"<<endl;
    }
private:
};

// 从“人”派生两个类，分别表示老师和学生
class Teacher : public Human
{
};

class Student : public Human
{
};

// 在主函数中模拟上车买票的场景
int _tmain(int argc, _TCHAR* argv[])
{
    // 声明一个基类的指针
    Human* pPassenger = NULL;
    // 车上上来一位老师
    pPassenger = new Teacher();
    // 老师买票
    pPassenger->BuyTicket();
    delete pPassenger;

    // 车上上来一位学生
    pPassenger = new Student();
    // 学生买票
    pPassenger->BuyTicket();
    delete pPassenger;

    pPassenger = NULL;

    return 0;
}

```

在这段程序中，我们先定义了一个基类 Human，它有一个行为（函数）是买票（BuyTicket()）。然后定义了它的两个派生类 Teacher 和 Student，通过继承，这两个派生类也拥有了买票的行为。在主函数中，假设第一个上车的人是老师，创建一个老师对象，让 Human 类型的指针 pPassenger 来指代这个对象，再利用 pPassenger 指针调用 BuyTicket() 函数，完成买票动作。又假设第二个上车的人是学生，创建一个学生对象，还是让 pPassenger 指针指代这个对象并调用它的 BuyTicket() 函数完成买票动作。最后，程序的输出结果是：

```

人买票。
人买票。

```

细心的你一定已经注意到，老师和学生买票的动作是一样的，因为都没有定义自己的

BuyTicket()函数，但是借助继承机制都拥有了 BuyTicket()这样一个行为，对 BuyTicket()的调用实质上是调用基类 Human 类的 BuyTicket()函数。虽然继承机制可以让派生类轻松地拥有基类的行为，但是在现实生活中，这种直接得来的行为不一定能满足需要。例如，虽然 Teacher 类和 Student 类都是从 Human 类继承的，但是它们的买票方式可能是不同的，比如老师需要投币买票，而学生则可以刷卡买票。也就是说，同一个动作，可能基类和派生类以及各个派生类之间的实现方式是不同的。这就像老“龙王”有 9 个儿子，这 9 个儿子却各不相同。为了满足各个派生类对类的行为进行自定义的需要，C++提供了虚函数的机制。在基类的函数声明前加上 virtual 关键字，这个函数就成为虚函数。这样，如果派生类对虚函数重新定义，那么派生类的虚函数实现将覆盖基类对应的虚函数的实现。如果通过基类指针调用虚函数，那么将调用这个指针所指向的具体对象的虚函数，以此来替代基类的虚函数。这样就提供了一个机会让派生类可以根据自己的实际情况重新定义基类的函数以满足实际的需要。现在，可以用虚函数的机制来修改上面的例子，让它更加符合实际。

// 经过虚函数机制改写后的“上车买票”演示程序

```
// 定义 Human 类，提供公有接口
class Human
{
    // Human 类的行为
public:
    // 这里将 BuyTicket() 函数声明为虚函数，
    // 表示其派生类可以对这个虚函数进行重新定义以满足实际的需要
    virtual void BuyTicket()
    {
        cout<<"人买票。"<<endl;
    }
private:
};

// 在派生类中对虚函数进行重新定义
class Teacher : public Human
{
public:
    // 根据实际情况重新定义基类的虚函数
    virtual void BuyTicket()
    {
        cout<<"老师投币买票。"<<endl;
    }
};

class Student : public Human
{
public:
    // 根据实际情况重新定义基类的虚函数
    virtual void BuyTicket()
    {
        cout<<"学生刷卡买票。"<<endl;
    }
};
```

```
    }  
};
```

使用虚函数机制改写程序之后，虽然在主函数中使用的还是基类的指针，但是执行的却是基类指针所指代的实际的派生类对象的函数。例如第一个“pPassenger->BuyTicket()”语句，因为这时 pPassenger 指针指向的是一个 Teacher 对象，而实际上执行的是 Teacher 类中的 BuyTicket() 函数，所以最后输出了更加符合实际的结果：

老师投币买票。

学生刷卡买票。

这里我们注意到，Human 类已经实现了 BuyTicket() 函数，如果它的派生类 Teacher 或者 Student 不实现这个函数，那么通过派生类调用 BuyTicket() 函数，最终还是调用基类 Human 这个已经实现的函数。如果想强制派生类定义某个函数，则可以在基类中将这个函数声明为纯虚函数，也就是基类不实现这个虚函数，它的所有实现都留给派生类来完成。例如：

```
// 使用纯虚函数  
// 这样 Human 类就成为了一个抽象类，仅提供接口  
class Human  
{  
    // Human 类的行为  
public:  
    // 声明 BuyTicket() 函数为纯虚函数  
    // 它的实现留待派生类来完成  
    virtual void BuyTicket() = 0;  
private:  
};
```

当类中有纯虚函数时，这个类就成为了一个抽象类，它仅用于对外界提供公有接口。同时，因为这个类包含有尚未完工的纯虚函数，所以不能创建抽象类的具体对象。如果试图创建一个抽象类的对象，将产生一个编译错误。例如：

```
// 编译错误，不能创建抽象类的对象  
Human aHuman;
```

如果某个类从抽象类派生，那么它必须实现其中的纯虚函数才能成为一个实体类，否则它将继续保持抽象类的特征。例如：

```
class Student : public Human  
{  
public:  
    // 实现基类中的纯虚函数，让 Student 类成为一个实体类  
    virtual void BuyTicket()  
    {  
        cout<<"学生刷卡买票。"<<endl;  
    }  
};
```

面向对象的多态机制为派生类修改基类的行为，以满足实际情况的需要提供了一种可能。

利用多态机制，可以为程序开发带来很多好处。

1. 接口统一，高度复用

应用程序不必为每个派生类编写具体的函数调用，只需要在基类中定义好接口就可，而具体实现再留给派生类自己去处理。这样就可以“以不变应万变”，大大提高了程序的可复用性（针对接口的复用）。

2. 向后兼容，灵活扩展

派生类的行为可以被基类的指针访问，可以很大程度上提高程序的可扩展性，因为一个基类的派生类可以很多，并且可以不断扩充。比如要增加一种乘客类型，只需要添加一个 Human 的派生类，实现自己的功能就可以了。在使用这个新创建的类的时候，无须修改程序代码中的调用形式。

6.4 实战面向对象：工资管理系统

在学习了面向对象思想及类之后，现在分析问题和解决问题似乎都戴上了“面向对象”这副有色眼镜，看一切事物都成了对象。什么？这么奇妙的眼镜你还没有体验过？来来来，快跟我一起来体验戴上“面向对象”这副有色眼镜看问题的奇妙感觉吧。

让我们戴上“面向对象”的有色眼镜来看这样一个熟悉的问题：要创建一个工资管理系统，该系统可以输入员工的姓名和入职时间。同时，职员分两个级别——经理和普通员工，两者的工资都根据入职时间计算，但是计算的方式不同。其中，经理的工资等于基本工资 10 000 元加上入职时间乘以 5 000 元，也就是每年工资递增 5 000 元；而普通员工的工资只是基本工资 2 000 元加上入职时间乘以 200 元。完成工资信息的录入后，系统就可以显示当前所有员工的信息并能够计算出所有员工的平均工资。

戴上“面向对象”的有色眼镜，在这个问题中我们首先看到的是什么？没错，是对象。

6.4.1 从问题描述中发现对象

问题描述中哪里有对象？我怎么没有看到？仔细看，问题描述中的各名词实际上就是对象。

这下豁然开朗了，我们来看看问题描述中有哪些名词，也就是要寻找的对象。首先，遇到的第一个名词是工资管理系统。然后是该系统所管理的员工，因为级别的不同，员工又分为经理和普通员工，这些就构成了整个问题中的所有对象。

除了找到对象之外，还可以发现对象之间的各种关系：工资管理系统管理员工对象，它们之间是一对多的关系；同时，经理和普通员工同属于员工，它们是从员工所派生出来的。

图 6-13 描述了整个问题中的对象及对象之间的关系。

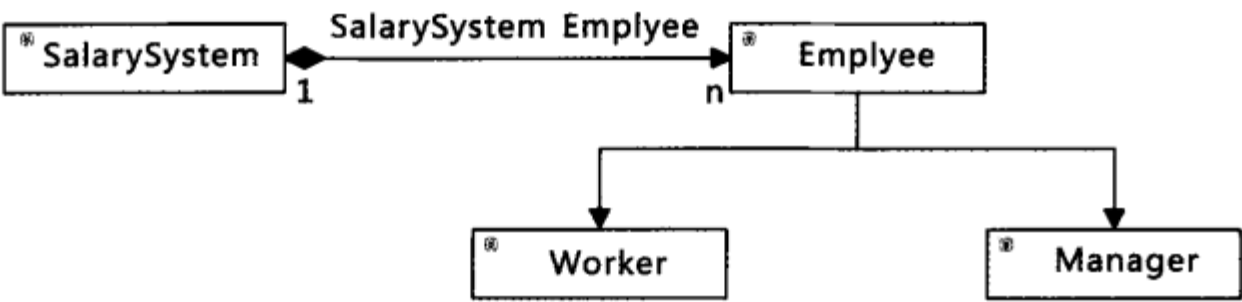


图 6-13 对象及对象之间的关系

6.4.2 分析对象的属性和行为

找到对象之后，就可以分析这些对象所拥有的属性和行为，利用面向对象封装的机制将其封装成具体的类。首先，分析这个问题中最基础的员工对象。根据问题描述，我们需要输入员工的姓名和入职时间，所以这个对象必需的属性是姓名和入职时间。同时，工资管理系统要显示所有员工的信息，这也就是员工对象必须对外提供公有接口以便外界能够获得它的姓名和入职时间属性。其次，根据员工级别的不同，员工被分成经理和普通员工两类，根据面向对象中继承的思想，为了复用基类的设计和代码，我们让这两个类从基类中派生，从而直接拥有基类的属性和行为。但是，在问题描述中，经理和普通员工这两个对象的工资计算方式是不同的，所以这里可以利用面向对象的多态机制对基类中的行为进行重新定义。通过这样的分析，我们就清楚了对象的属性和行为，利用面向对象的多态机制将这些属性和行为封装起来，就形成了具体的类，如图 6-14 所示。

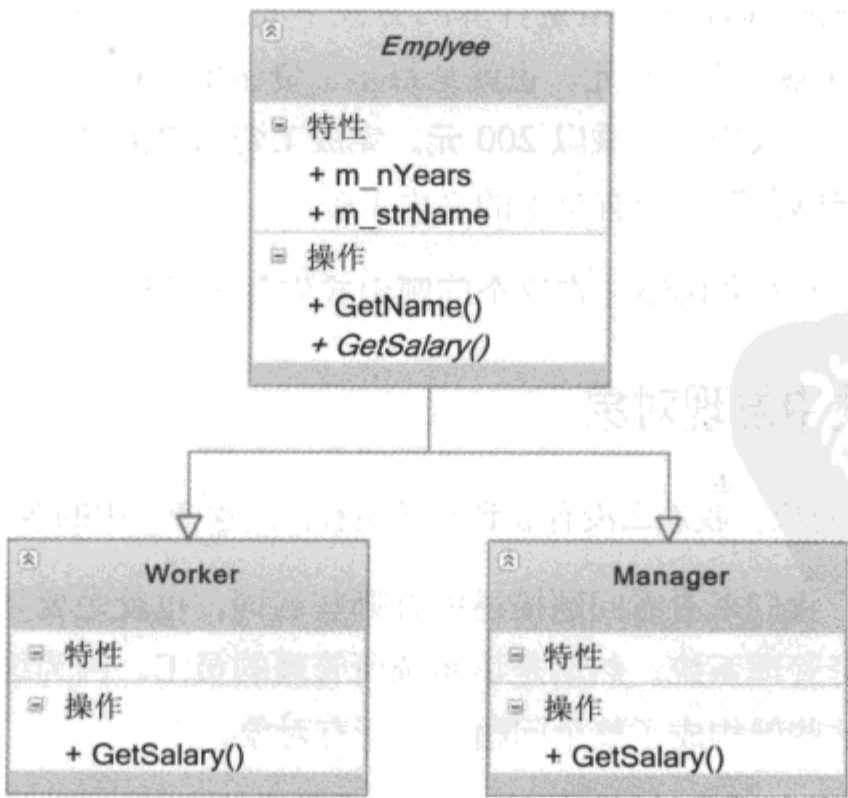


图 6-14 员工类的属性和行为

完成了员工相关类的创建后，我们来看看对员工对象进行管理的工资管理系统，分析它应该具有哪些属性和行为。根据问题描述，工资管理系统要对所有输入的员工对象进行管理，所以它必须有一个属性来保存它所管理的所有员工并且有一个属性用来记录当前的员工总数。因为要管理多个员工对象，所以可以使用数组作为其保存员工对象的属性。为了处理问题描述中的事务，例如员工信息的输入和显示及计算平均工资等，它必须具备相应的行为，也就是它必须提供相应的成员函数供外界调用以完成相应的事务。经过这样的分析，工资管理系统类实现如图 6-15 所示。

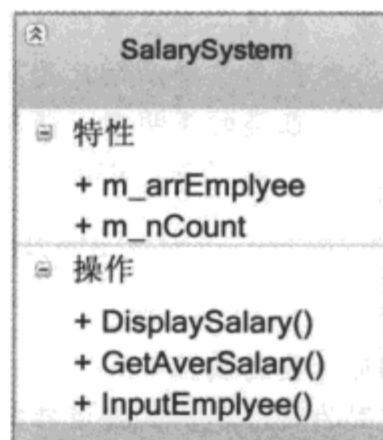


图 6-15 工资管理系统类的属性和行为

6.4.3 实现类的属性和行为

如果你是架构师，只要你完成了类的设计，那么你的工作就可以到此结束了。但如果是程序员，那么还需要将类的设计用 C++ 语言来实现。

首先，按照 6.4.2 小节中对员工类及从它派生的经理类和普通员工类的设计，可以将这三个类实现如下：

```
// SalarySystem.h 头文件
// 用于类的声明
#pragma once

// 引入需要用到的头文件
#include <string>
#include <iostream>
using namespace std;

// 员工类
class Employee
{
public:
    // 构造函数
    // 根据员工的名字和入职时间构造员工对象
    Employee( string strName, int nYears )
        : m_strName( strName ),
          m_nYears( nYears )    // 利用构造函数参数对属性进行初始化
    {};

    // 员工类的行为
public:
    // 提供一个纯虚函数作为公有接口，供外界获得员工的工资
    // 因为经理类和普通员工类这两个派生类对工资的计算方式
```

```

// 不同，所以这里必须提供纯虚函数供派生类对其进行自定义
virtual int GetSalary() = 0;

// 提供一个公有接口，供外界得到员工的名字
// 因为这个函数只是简单地返回一个属性值，所以把它定义
// 在类的声明中，使其成为一个内联函数，提高性能
string GetName()
{
    return m_strName;
};

// 员工类的属性
// 因为这些属性需要遗传给它的派生类，
// 所以这里将其访问级别设置为 protected
protected:
    // 入职时间
    int m_nYears;
    // 姓名
    string m_strName;
};

// 经理类
// 因为经理是员工中的一种，所以它从员工类 Employee 派生
class Manager : public Employee
{
public:
    // 使用基类的构造函数，完成对属性的初始化工作
    Manager(string strName, int nYears )
        : Employee( strName, nYears )
    {};
public:
    // 对基类提供的纯虚函数进行自定义。
    // 根据需求描述我们使用了不同的工资计算方式，利用面向对象
    // 的多态机制，使得同名的函数在不同的派生类中可以有不同的实现。
    virtual int GetSalary()
    {
        return 5000 * m_nYears + 10000;
    }
};

// 普通员工类
class Worker : public Employee
{
public:
    Worker(string strName, int nYears )
        : Employee(strName, nYears )
    {};
public:
    // 对基类的纯虚函数进行自定义
    virtual int GetSalary()
    {
        return 200 * m_nYears + 2000;
    }
};

```

```
};
```

在员工类及其派生类的实现中，全面体现了面向对象的三大特征。首先，我们将所有员工，包括经理和普通员工的共有属性和行为封装成员工类 Employee 这个基类；然后使用面向对象的继承机制从员工类 Employee 中派生出经理类 Manager 和普通员工类 Worker，这样使得这两个派生类可以复用基类的代码，无须重复实现基类已有的属性和行为，例如员工的姓名和入职时间等共有属性，以及供外界访问的 GetName()等接口函数。派生类所要做的只是实现自己特有的属性和行为。例如，两个派生类各自对工资的计算方式不同，所以利用面向对象多态的机制，它们对基类提供的虚函数进行自定义，各自完成自己的 GetSalary()这一函数的定义。

完成员工类及其派生类之后，就可以在工资管理系统类 SalarySystem 中创建对象并使用这些对象来代表员工，对工资进行管理。SalarySystem 实现如下：

```
// SalarySystem.h 头文件

// 定义一个常量表示最大员工数
const int MAX_COUNT = 1000;

// 工资管理系统
class SalarySystem
{
public:
    // 构造函数和析构函数
    SalarySystem(void);
    ~SalarySystem(void);
    // 工资管理系统的行为
public:
    // 输入员工信息
    void InputEmployee(void);
    // 显示员工信息
    void DisplaySalary(void);
    // 计算获得平均工资
    double GetAverSalary(void);
    // 工资管理系统的属性
    // 因为这些属性仅仅供工资管理系统自己使用，所以我们将其
    // 访问级别设置为 private
private:
    // 当前员工总数
    int m_nCount;
    // 用于保存所有员工对象
    // 这里将员工对象的指针保存到数组中，通过指针访问员工对象
    Employee* m_arrEmployee[MAX_COUNT];
};
```

因为 SalarySystem 的各个成员函数的实现比较复杂，所以我们将其实现放到跟.h 头文件

相对应的.cpp 源文件中进行：

```
#include "StdAfx.h"
// 引入类定义所在的头文件
#include "SalarySystem.h"

// 构造函数，对类的属性进行初始化
SalarySystem::SalarySystem(void)
{
    m_nCount = 0;    // 将员工总数初始化为 0
}

// 析构函数、清理资源、释放内存
SalarySystem::~SalarySystem(void)
{
    // 循环遍历保存员工对象指针的数组，清理资源
    for( int i = 0; i < m_nCount; ++i )
    {
        Employee* pEmployee = m_arrEmployee[i];
        // 销毁对象，释放内存
        delete pEmployee;
        // 将相应指针设置为 NULL，不可访问
        m_arrEmployee[i] = NULL;
    }
}

// 获取用户输入
void SalarySystem::InputEmployee(void)
{
    // 显示提示信息
    cout<<"请输入员工信息\n"<<
        "格式：员工姓名 入职时间 是否为经理级别\n"<<
        "例如：ChenLiangqiao 4 0\n"<<
        "输入 end 表示输入结束"<<endl;
    // 局部变量，用于接收用户输入
    string strName = "";
    int nYears = 0;
    bool bManager = false;
    int nIndex = 0;

    // 开始循环接收用户输入的用户数据
    while( nIndex < MAX_COUNT )
    {
        // 清空输入流
        cin.clear();
        // 从输入流读取用户输入的数据
        cin>>strName>>nYears>>bManager;
        // 判断是否输入结束，如果结束，则跳出循环
        if( "end" == strName )
            break;
        // 根据用户输入创建相应的员工对象
        Employee* pEmployee = NULL;
```

```

        if( bManager )
        {
            // 如果用户输入的是一个经理，则创建 Manager 对象
            pEmployee = new Manager(strName, nYears);
        }
        else
        {
            // 如果用户输入的是一个普通员工，则创建 Worker 对象
            pEmployee = new Worker(strName, nYears);
        }
        // 将创建的员工对象指针保存到数组中
        m_arrEmployee[ nIndex ] = pEmployee;
        // 索引值递增，开始下一次录入循环
        ++nIndex;
    }

    // 保存输入的员工总数
    m_nCount = nIndex;
}

// 显示输出工资信息
void SalarySystem::DisplaySalary(void)
{
    // 显示工资信息
    cout<<"工资管理系统"<<endl;
    cout<<"当前员工总数: "<<m_nCount<<
        "\n 平均工资是: "<<GetAverSalary()<<endl;
    cout<<"员工具体工资信息如下: "<<endl;
    // 循环遍历保存员工对象指针的数组，输出每个员工对象的具体信息
    // 这里使用的是基类指针调用基类的公有接口函数，但是如果这个
    // 接口函数是虚函数，那么它们将调用这个指针所指向的实际对象的相
    // 应函数。例如，这里对 GetSalary() 函数的调用，编译器将根据 pEmployee
    // 这个基类指针所指向的具体对象是 Manager 还是 Worker 来决定到底是
    // 调用 Manager 类中的 GetSalary() 实现还是 Worker 类中的 GetSalary() 实现。
    // 这就是面向对象中多态的体现。
    for( int i = 0; i < m_nCount; ++i )
    {
        Employee* pEmployee = m_arrEmployee[i];
        cout<<pEmployee->GetName()<<"\t"<<
            pEmployee->GetSalary()<<endl;
    }
}

// 计算平均工资
double SalarySystem::GetAverSalary()
{
    int nTotal = 0;
    // 计算工资总额
    for( int i = 0; i < m_nCount; ++i )
    {
        Employee* pEmployee = m_arrEmployee[i];
        nTotal += pEmployee->GetSalary();
    }
}

```



```

    }

    // 返回平均工资
    return (double)nTotal / (m_nCount);
}

```

完成工资管理系统类 SalarySystem 之后，已经是万事俱备，只欠东风了。我们只需在主函数中简单地使用这个类就完成了整个系统。

```

#include "stdafx.h"
// 引入 SalarySystem 声明所在的头文件
#include "SalarySystem.h"

// 在主函数中使用 SalarySystem 对象
int _tmain(int argc, _TCHAR* argv[])
{
    // 创建 SalarySystem 对象
    SalarySystem nSalarySys;
    // 获取用户输入
    nSalarySys.InputEmployee();
    // 显示具体的工资信息
    nSalarySys.DisplaySalary();

    return 0;
}

```

有了类的帮助，短短几行代码就能完成整个系统。面向对象思想更加接近我们思考问题、解决问题的思维模式，这使得工资管理系统的设计更加直观、更加贴近现实的需求。在功能上，它比以前更强大，程序结构和代码比以前更加简单。同时，整个系统也更加容易理解，使得它具有很好的可扩展性和可维护性。这就是面向对象的魅力所在。

面向对象分析方法

这里只是利用面向对象思想对一个小问题进行了分析。当利用面向对象思想设计开发比较大的系统时，过程当然不会这么简单。实际上，面向对象开发方法的研究已日趋成熟，国际上已有不少面向对象开发的方法出现。

1. Booch 方法

Booch 方法最先描述了面向对象软件开发方法的基础问题，指出面向对象开发是一种不同于传统的功能分解的设计方法。面向对象的软件分解更接近人对客观事务的理解，而功能分解只能通过问题空间的转换来获得。

2. Coad 方法

Coad 方法是由 Coad 和 Yourdon 于 1989 年提出的面向对象开发方法。该方法的主要优点是通

提出的一套系统的原则。该方法完成了从需求角度来进一步进行类和类层次结构的认定。尽管 Coad 方法没有引入类和类层次结构的术语，但事实上已经在分类结构、属性、操作、消息关联等概念中体现了类和类层次结构的特征。

3. OMT 方法

OMT 方法是由 James Rumbaugh 等 5 人于 1991 年提出来的。该方法是一种新兴的面向对象的开发方法，开发工作的基础是对现实世界的对象建模，并围绕这些对象使用分析模型来进行独立于语言的设计。面向对象的建模和设计促进了对需求的理解，有利于开发更清晰、更易维护的软件系统。该方法为大多数应用领域的软件开发提供了一种实际的、有效的保证。

4. UML (unified modeling language) 语言

UML 是面向对象技术领域内占主导地位的标准建模语言。它不仅统一了 Booch 方法、OMT 方法和 OOSE 方法的表达系统结构的方式，而且对它们有了更进一步的发展，最终统一为大众接受的标准建模语言。UML 是一种定义良好、易于表达、功能强大且普遍适用的建模语言。更多地，它以图形的方式来描述整个系统以及系统中各个组成部分动态或者静态的关系。它的作用域不限于支持面向对象的分析与设计，还支持从需求分析开始的软件开发全过程。

本章中使用的各种描述工资管理系统的图，就是 UML 系列图形中的一部分。

6.5 高手是这样炼成的

C++ 中的类和面向对象如此博大精深，让无数高手竞折腰！

6.5.1 C++ 类对象的内存模型

要获得自由，必须知道事情的真相，而关于 C++ 类对象的全部真相，就在它的内存模型当中。

类是对属性和行为的封装，在类的对象中也应该有属性（成员变量）和行为（成员函数）。反映到类对象的内存模型中，也就是内存中应该有对象的成员变量和成员函数。这样描述是不是很抽象？现在来学学庖丁，将一个类对象的内存模型进行解剖，看看其中到底是什么东西。例如，有这样一个类：

```
class Base
{
    // 行为
public:
    void foo1(void){};
    void foo2(void){};
    // 属性
private:
```

```

        double m_fMember1;
        int m_nMember2;
};

```

其中，Base 类有两个成员变量 m_fMember1 和 m_nMember2，以及两个成员函数 foo1()和 foo2()。现在就将这个类的对象“大卸八块”，看看这些成员变量和成员函数在内存中到底是如何分布的。

```

// 定义类成员函数指针类型
// 用于得到类的成员函数指针
typedef void (Base::*CLASS_FUNC)(void);

int _tmain(int argc, _TCHAR* argv[])
{
    // 创建一个 Base 类的对象
    Base base;
    // 输出类对象成员变量的地址
    cout<<"类对象的地址是: "<<&base<<endl;
    cout<<"类对象成员变量 m_fMember1 的地址是: "<<
        &(base.m_fMember1)<<endl;
    cout<<"类对象成员变量 m_fMember1 占用的内存字节数是: "<<
        sizeof(double)<<endl;
    cout<<"类对象成员变量 m_nMember2 的地址是: "<<
        &(base.m_nMember2)<<endl;

    // 输出类成员函数的地址
    // 第一个函数
    CLASS_FUNC pFunc = &Base::foo1;
    unsigned* tmp = (unsigned*)&pFunc;
    cout<<"Base 类第一个成员函数的地址是: "<<hex<<*tmp<<endl;
    // 第二个函数
    pFunc = &Base::foo2;
    tmp = (unsigned*)&pFunc;
    cout<<"Base 类第二个成员函数的地址是: "<<hex<<*tmp<<endl;

    return 0;
}

```

运行这个程序，类对象的检测报告如下。

```

类对象的地址是: 001CF820
类对象成员变量 m_fMember1 的地址是: 001CF820
类对象成员变量 m_fMember1 占用的内存字节数是: 8
类对象成员变量 m_nMember2 的地址是: 001CF828
Base 类第一个成员函数的地址是: 41217
Base 类第二个成员函数的地址是: 410e6

```

仔细分析这份类对象的检测报告就会发现，对象的第一个成员变量的地址跟整个对象的地址相同，这表明在对象的内存模型中排在第一位的就是第一个成员变量。第二个成员变量的地

址就是第一个成员变量的地址加上它所占用的内存空间，换句话说，第二个成员变量紧排在第一个成员变量之后。由此可以看出，对象中的成员变量是按照类声明中的顺序依次排列的。

既然成员变量是依次排列的，那么成员函数会不会也如此呢？看看检测报告中的成员函数的地址，发现成员函数的地址是一个奇怪的内存地址。这是为什么呢？原来跟每个对象都有一份成员变量不同，并不是每个对象都有一份成员函数。因为同一个类的所有对象的成员函数都是相同的，没有必要为每个对象都配备一份成员函数。在 C++ 类对象模型中，类的所有成员函数都被放在一个特殊的位置，所有这个类的对象都共用这份成员函数。整个类对象的内存模型如图 6-16 所示。

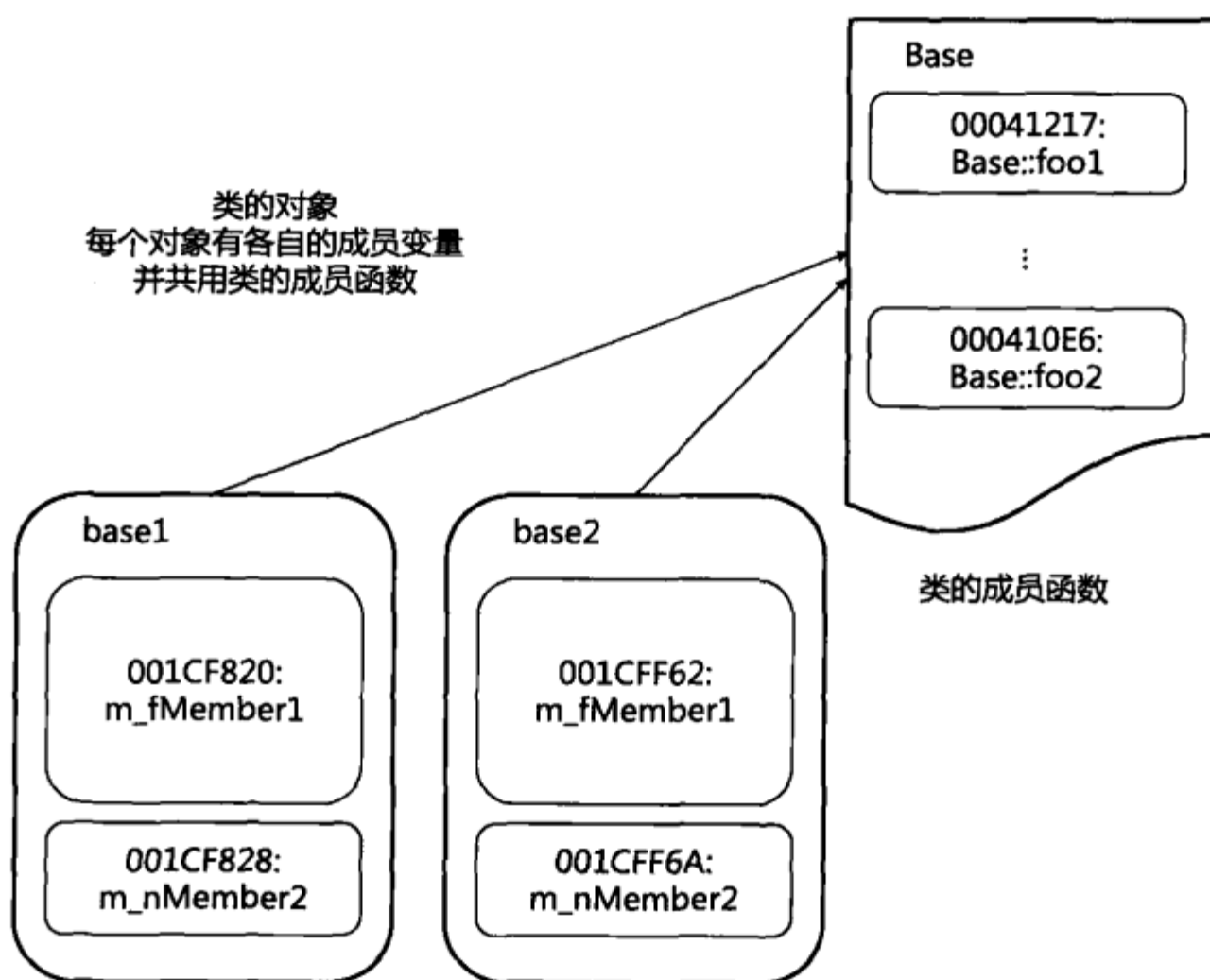


图 6-16 类对象的内存模型

如果类中有虚函数存在，则情况会发生一点点变化。为了让对象能够调用它的虚函数，在每个对象最开始的内存位置添加一个虚函数表的指针_vfptr，其后才是对象的成员变量内存数据。如果某个类是个派生类，那么它的对象内存中最开始的地方实际上是基类对象的拷贝，包括基类虚函数表指针和成员变量，其后才是派生类自己的成员变量数据。

6.5.2 指向自身的 this 指针

聪明的你在学过类对象的内存模型之后，一定会问这样一个问题：既然各个对象都使用

同样一份成员函数代码，那么在成员函数之中，如果要对各个具体对象的成员变量进行访问，又如何找到这个成员变量所属的具体对象呢？例如，有这样一个简单的类：

```
class Base
{
public:
    // 成员函数访问成员变量
    void SetValue( int nVal )
    {
        m_nVal = nVal;
    }
    // 类的成员变量
private:
    int m_nVal;
};
```

在这个类的 SetValue()成员函数中，我们利用参数对成员变量进行赋值。现在就创建相应的对象调用其成员函数：

```
// 创建对象
Base aBase;
// 调用成员函数对成员变量进行赋值
aBase.SetValue( 1 );
```

当需要访问类的成员变量时，总是要指明该类的一个具体实例对象，编译器才会知道它要访问的是哪个对象的成员变量，该成员变量在内存中的什么地方。可是经过仔细观察代码发现，SetValue()函数中并没有指明 m_nVal 成员变量到底属于哪一个对象。那么“m_nVal = nVal”赋值语句又将数值赋值给了谁呢？m_nVal 变量一直在问：Who am I？代码后面一定隐藏了秘密，我们要知道真相！

是的，编译器隐藏了 C++ 中的一位特殊人物——this 指针。this 是 C++ 中的一个关键字，它代表指向当前对象的指针，而成员函数中的成员变量就是通过 this 指针找到自己所属的对象的。在成员函数中，如果将隐藏的 this 指针添加上去，整个世界就豁然开朗了。

```
class Base
{
public:
    // 成员函数访问成员变量
    void SetValue( int nVal )
    {
        // 显示使用 this 指针访问成员变量
        this->m_nVal = nVal;
    }
    // ...
};
```

从以上代码中可以清楚地看到，类成员函数中的成员变量前都有一个 this 指针。当通过某个对象调用它的成员函数时，系统会隐式地传递给成员函数一个指向这个对象的指针，这

就是 this 指针。当在成员函数中访问类的成员变量时，这时 this 指针指向的是调用这个函数的对象，所以对成员变量的访问也就变成对这个对象所属的成员变量的访问。例如，通过 aBase 对象调用 SetValue() 成员函数，那么在 SetValue() 成员函数中，隐藏的 this 指针就指向 aBase 这个对象。显然，成员函数中的 “this->m_nVal = nVal” 语句也就是对 aBase 的 m_nVal 成员变量赋值，如图 6-17 所示。

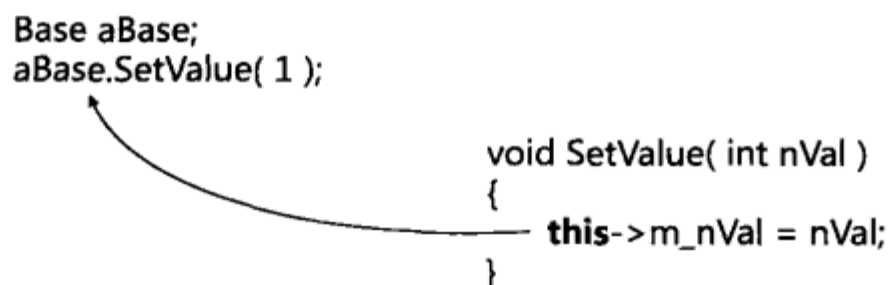


图 6-17 this 指针

事实上，类的每个非静态成员函数中都隐式地声明了 this 指针，该指针指向调用该成员函数的对象。利用该指针，可以访问到该对象的所有成员变量和成员函数，例如：

```
class Base
{
public:
    // 成员函数访问成员变量
    int ChangeValue( int nVal )
    {
        // 通过 this 指针访问成员函数
        this->SetValue( nVal )
        // 通过 this 指针访问成员变量
        return this->m_nVal;
    }
    // ...
};
```

在实际开发中这样做的意义并不大，因为编译器在编译代码的时候会自动给成员函数中对类成员的访问加上 this 指针。更多时候，this 指针用来返回指向对象本身的指针以实现对象的链式引用，或者避免对同一对象进行赋值操作。例如：

```
// 描述一个点位置的类
class Point
{
public:
    Point( int x, int y )
        : m_nX(x), m_nY(y)
    {};
    // 重载赋值操作符 “=”，进行赋值操作
    void operator = (Point& pt)
    {
        // 判断传递进来的参数是否是这个对象自身
        // 如果是同一对象，则不进行赋值操作
    }
};
```

```

        if( this != &pt )
        {
            m_nX = pt.m_nX;
            m_nY = pt.m_nY;
        }
    }
    // 移动点的位置
    Point& Move( int x, int y )
    {
        m_nX += x;
        m_nY += y;
        // 返回对象本身，这样可以利用函数返回值进行链式引用
        return *this;
    }
private:
    int m_nX;
    int m_nY;
};

```

使用 this 指针之后，Point 类腰也不酸了，背也不疼了，爬坡更有劲了。

```

Point pt1(2,4);
Point pt2(0,0);
// 自己赋值给自己，这不明摆着瞎耽误工夫吗？
// this 指针是不答应的
pt1 = pt1;
// 移动一下，再移动一下
// 通过返回对象本身实现对象的链式引用
pt1.Move(1,1).Move(2,4);

```



C++世界的奇人异事

在武侠小说中，初入武林的毛头小子总是要遇到几位奇人，发生几件异事。经过高人的指点，经历一番磨炼，方能武功精进，从一个新手成长为高手。在 C++ 世界，同样有诸多的奇人异事。在 C++ 世界中游历学习的我们，是否也期望遇到几位奇人，经历几件异事，而从一个 C++ 新手成长为 C++ 高手呢？

武林中的奇人异事可遇而不可求，但是 C++ 世界中的奇人异事却可以为你一一引见。

7.1 一切指针都是纸老虎：彻底理解指针

C++ 世界中什么最难？指针！C++ 世界中什么最威力无边？指针！

指针作为 C++ 世界中所特有的一种数据处理方式，因为其使用方式的灵活而给 C++ 世界带来了无比的威力；然而，也正是因为它的灵活，使得指针成为初学者最难掌握的 C++ 技能。它就像一只吊睛白额“大老虎”，虽然凶猛无比、牙尖嘴利、不容易掌控，但可以帮助我们灵活地访问内存，解决很多问题。今天就来打倒指针这只“纸老虎”，彻底理解指针。

7.1.1 指针的运算

虽然指针所表示的意义比较特殊，但是从本质上讲，指针首先是一种基本数据类型，可以参与部分运算，包括算术运算、关系运算和赋值运算。我们最常用的就是指针的算术加减运算。

指针的本质是内存的地址，它指向某一个内存位置。指针的加减运算，实际上是让指针发生偏转，指向另外的内存位置。通过这种指针的偏转，可以访问到该指针附近的内存。例如，在 3.6 节中介绍过数组，数组名实际上就是数组的首地址，表示数组在内存中的起始位置。可以通过把首地址赋值给指针，然后对该指针进行加减运算，使指针发生偏转指向数组中的其他元素，从而遍历整个数组。例如：

```
int nArray[3] = { 1, 2, 3 }; // 定义一个数组
int* pIndex = nArray;       // 将数组的起始地址赋值给指针 pIndex
```

```

cout<<"指针指向的地址是："<<pIndex<<endl;          // 输出指针指向的地址
cout<<"指针所指向的数据的值是："<<*pIndex<<endl; // 输出值

pIndex++;                      // 对指针进行加运算，使其指向数组中的下一个值

cout<<"指针指向的地址是："<<pIndex<<endl;          // 输出指针指向的地址
cout<<"指针所指向的数据的值是："<<*pIndex<<endl; // 输出值

```

这段程序执行后，可以得到这样的输出：

```

指针指向的地址是：0016FA38
指针所指向的数据的值是：1
指针指向的地址是：0016FA3C
指针所指向的数据的值是：2

```

从输出结果中可以看到这个指针初始指向的地址是 0016FA38，也就是 nArray 这个数组的首地址。在对指针进行加 1 运算后，指针指向的地址变为 0016FA3C，它向地址增大的方向偏移了 4 个单位，指向了数组中的第二个数。

大家肯定会奇怪，对指针进行的是加 1 的运算，怎么指针指向的地址却增加了 4 个单位？这是因为指针跟它所指向的数据的真正数据类型相关，指针加 1 或者减 1，会使指针指向的地址增加或者减少一个对应的数据类型的长度。比如以上代码中的 int 型指针，它的加 1 运算就使地址增加了 4 个单位，也就是一个整型数的长度。同理，对字符型指针加 1，地址实际增加 1；对双精度型指针加 2，地址实际增加 16。指针偏转流程如图 7-1 所示。

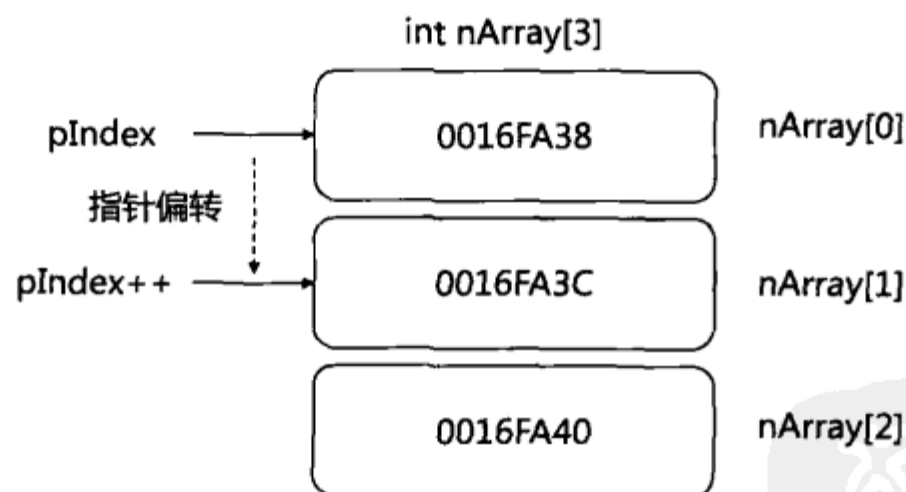


图 7-1 指针偏转

除了指针的加减算术运算之外，常用到的还有指针的关系运算。指针的关系运算通常用来判断两个相同类型的指针是否相等，是否指向同一个地址。例如：

```

int nArray[3] = { 1, 2, 3 }; // 定义一个数组
int* pIndex = nArray;        // 将数组的起始地址赋值给指针 pIndex
int* pEnd = nArray + 3;      // 计算数组的结束地址
while( pIndex != pEnd )      // 判断当前指针是否已经偏转到结束地址
{

```

```

        cout<<*pIndex<<endl;           // 输出当前数据
        ++pIndex;                       // 偏移到下一个内存位置
    }

```

在以上这段代码中，利用当前指针跟表示数组结束位置的指针进行比较来判断当前指针是否已经偏转到数组的结束位置，如果已经到达结束位置就结束循环。另外，指针变量也常和 NULL 宏进行比较，来判断指针是否被初始化而指向正确的内存位置，也就是判断这个指针是否有效。有时在声明一个指针时，并没有合适的初始值赋给它，但如果让它保存最开始的随机值，又会产生不可预见的结果。在这种情况下，我们会在声明这个指针后，就将这个指针赋值为 NULL，表示这个指针还没有被初始化，处于不可用的状态。等到合适的时候，再将真正有意义的值赋值给它来完成这个指针的初始化。将 NULL 跟指针进行比较，是判断一个指针是否可用的常用手段。下面是一个典型的应用。

```

int* pInt;           // 定义一个指针，这时指针是一个随机值，指向随机的一个内存地址
pInt = NULL;         // 将指针赋值为 NULL，表示指针还没有合适的值，不可用

//...

int nArray[10];
pInt = nArray;       // 将数组首地址赋值给指针
if( NULL != pInt ) // 判断指针是否初始化完成
{
    // 指针有效，可以开始使用指针
}

```

将指针与 NULL 进行比较，可以有效地避免指针的非法访问，这是一条非常好的编程经验。

知道更多：什么是 NULL

NULL 实际上是一个宏，在 C++ 语言中，它的定义是：

```
#define NULL 0
```

可以看到，NULL 实质上就是整数 0。0 值具有特殊性，我们常常用 NULL 值表示一个指针变量是一个空指针，它并没有指向一个正确的内存地址，而是一个无效的不可访问的指针。

除了 NULL 宏之外，还可以使用 nullptr 关键字来表示一个空指针，例如：

```

// 定义一个指针，并将它赋值为空指针
// 表示此时指针不可用
int* pInt = nullptr;
// 判断指针是否为空指针
// 如果是空指针，就对其进行赋值
if ( nullptr == pInt )
{

```

```
pInt = &nArray; // 将数组的首地址赋值给指针
}
```

7.1.2 灵活的 void 类型和 void 类型指针

C++是一种强类型的语言，其中的数据可分为不同类型，例如整型数、字符型数等。但是，在C++世界中却出现了一个异类，那就是void类型。从本质上讲，void类型并不是一个真正的数据类型，它并不能定义一个void类型的数据，因为这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人。void更多的是体现一种抽象，在程序中，void类型更多的是用于“修饰”和“限制”一个函数。例如：如果一个函数没有返回值，则可用void作为这个函数的返回值类型，表示这个函数没有返回值；如果一个函数没有形式参数列表，也可用void作为其形式参数，表示这个函数不需要任何参数。

跟void类型的“修饰”作用不同，void类型指针作为指向抽象数据的指针，它可以成为两个具有特定类型指针之间相互转换的桥梁。众所周知，如果两个指针的类型相同，那么可以直接在这两个指针之间互相赋值；如果两个指针指向不同的数据类型，则必须使用强制类型转换运算符，把赋值操作符右边的指针类型转换为左边的指针类型。例如：

```
int* pInt;           // 指向整型数的指针
float* pFloat;       // 指向浮点数的指针
pInt = pFloat;       // 直接赋值会产生编译错误
pInt = (int*)pFloat; // 强制类型转换后进行赋值
```

但是，当使用void类型指针时，就没有类型转换的麻烦。void类型指针显得八面玲珑，任何其他类型的指针都可以直接赋值给void类型指针，例如：

```
void* pVoid;         // void类型指针
int* pInt;           // int类型指针
pVoid = pInt;        // 任何其他类型的指针都可以直接赋值给void类型指针
```

这并不意味着void类型指针可以无须强制类型转换而直接赋值给其他类型的指针。因为“无类型”可以包容“有类型”，而“有类型”却不能包容“无类型”。道理很简单，就像我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”一样。要将一个void类型指针赋值给其他类型指针，必须经过强制类型转换，让“无类型”变成“有类型”。例如：

```
void* pVoid;         // void类型指针
float* pFloat;       // float类型指针
pFloat = (float*)pVoid; // 通过强制类型转换，将void类型指针转换成float类型指针
```

当然，如果把void类型指针转换成并不是它实际指向的数据类型，其结果是不可预测的。比如把int类型指针赋值给void类型指针，然后把void类型指针强制转换成double类型指针

并间接引用它，这样的结果将不可预测。所以使用时应该把 void 类型指针转换成它实际指向的类型。

因为 void 类型指针对所指向的内存数据类型并没有要求，所以它可以用来代表任何类型的指针，如果函数可以接受任何类型的指针，那么应该将其参数声明为 void 类型指针。例如内存复制函数：

```
void * memcpy(void *dest, const void *src, size_t len);
```

在这里，任何类型的指针都可以传入 memcpy 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么数据类型。如果 memcpy() 函数的参数类型不是 void 类型指针，而是 char 类型指针或者其他类型指针，那么在调用 memcpy() 函数时，就需要不断地进行指针类型的转换，纠缠于具体的数据类型，这样的 memcpy() 函数明显不是一个“纯粹的、脱离低级趣味的”内存复制函数。

最佳实践：指针类型的转换

虽然指针类型的转换可能会带来不可预料的麻烦，就像 C++ 世界中“臭名昭著”的 goto 语句一样，但是在某些时候，指针类型的转换也可以发挥很大的作用，例如需要将某个指针转换成函数参数所要求的指针类型，以达到调用这个函数的目的时，指针类型的转换就成为一种必需。

在 C++ 世界中，可以使用 C 风格的强制类型转换进行指针类型的转换，也就是在指针前加上新的类型说明符将其转换成新的类型。例如：

```
int* pInt;  
float* pFloat = (float*)pInt; // 强制类型转换
```

这里，使用了 (float*) 这个新的指针类型说明符将 pInt 整型指针转换成了浮点型指针 pFloat。虽然这种方式比较直接，但是非常粗鲁，因为它允许你在任何类型之间进行转换。另外，这种类型的转换方式在程序语句中很难识别，代码阅读者可能会忽略类型转换的语句。

为了克服 C 风格类型转换的这些缺点，C++ 世界引进了新的类型转换操作符 static_cast。在大多数情况下，我们习惯使用这样的方式进行类型转换：

(类型说明符) 表达式

现在，使用 static_cast 应该写成这样：

```
static_cast<类型说明符>(表达式)
```

其中，表达式是已有的旧数据类型数据，而类型说明符就是要转换成的新数据类型的说明符。static_cast 在功能上与 C 风格的类型转换一样强大，含义也一样。例如，把一个 int 类型转换成 double 类型，以便让包含 int 类型变量的表达式产生出浮点数值的结果。如果用 C 风格的类型转换，可以这样写：

```
int nVal1,nVal2;  
double fResult = ((double)nVal1)/nVal2;
```

如果用 `static_cast` 进行类型转换，应该这样写：

```
double fRelult = static_cast<double>(nVal1)/nVal2;
```

使用 C++风格的类型转换，不论是对代码阅读者还是对程序都很容易识别。我们应该在代码中尽量避免进行类型转换，但如果类型转换无可避免，那么使用 C++风格的类型转换在一定程度上既可增加代码的可读性，也是对类型转换损失的一种补偿。

7.1.3 指向指针的指针

指针变量可以指向整型变量、字符型变量等基本数据类型的变量，也可以指向指针类型变量。当指针变量用于指向指针类型变量时，称为指向指针的指针变量。这句话虽然有些像绕口令，但其实可以这样理解：指针也是一个变量，在内存中的某个地址存放着这个变量，当有另外指针指向这个地址时，这个指针就是我们所说的指向指针的指针了。怎么？还是难以理解？没有关系，下面来看一个实际的例子：

```
int N = 2;  
int* pN = &N;  
int** ppN = &pN;
```

在这段代码中，首先定义了一个整型变量 `N`，然后定义了一个整型指针指向这个变量 `N`。换句话说，这个指针的值就是整型变量 `N` 在内存中的地址。最后，指向指针的指针登场了，我们用 `**` 定义了一个指向 `pN` 指针的指针，`ppN` 中保存的就是 `pN` 指针变量的地址，也就是它指向这个整型指针。图 7-2 展示了这三个变量之间的内存关系。

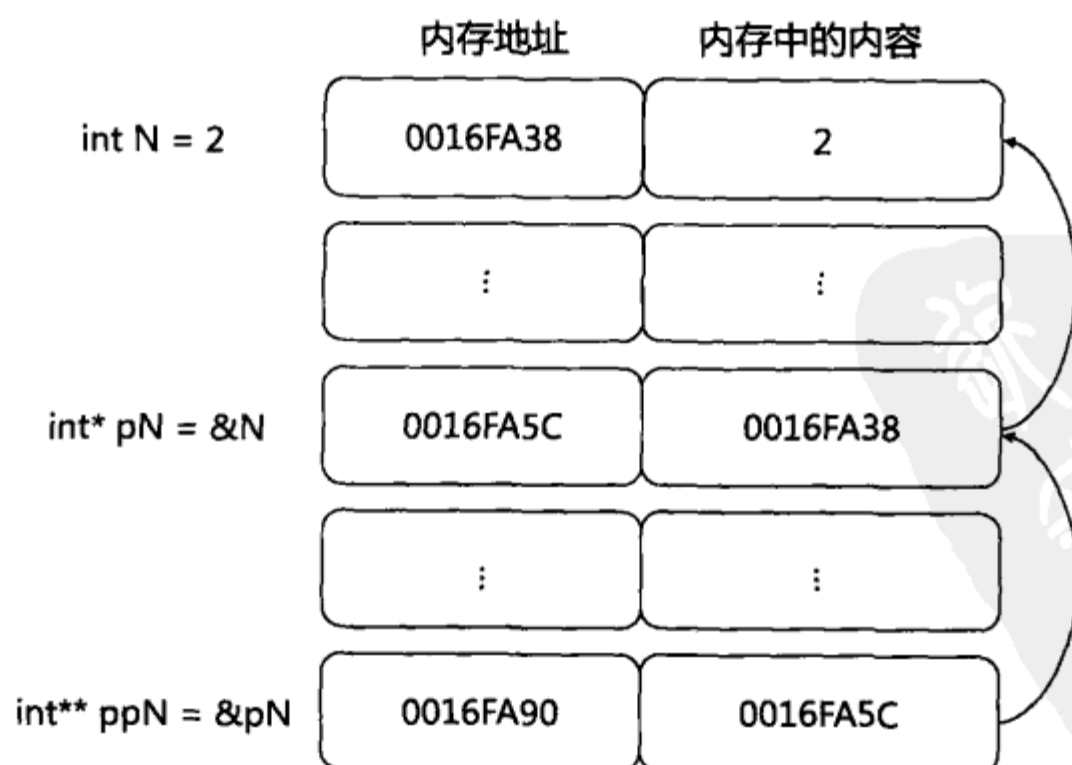


图 7-2 指向指针的指针

从图 7-2 中可以看到，变量 N 保存在 0016FA38 这个内存位置，而 pN 这个指针变量保存的是 N 所在的内存位置 0016FA38，同时，它自己保存在 0016FA5C 这个内存位置。同理，作为指向 pN 的指针，ppN 这个指针变量保存的是 pN 所在的内存位置 0016FA5C。简单来讲，普通指针指向的是一个具体的数据，而指针的指针所指向的是一个指针。

在 C++ 中，可以用下面的语法格式来声明一个指向指针的指针：

数据类型标识符** 指针变量名

其中，数据类型表示它所指向的指针的数据类型。例如：

```
int** ppN = &pN;
```

这样就定义了一个指向整型指针的指针 ppN，它指向另一个指针变量 pN，而 pN 指针变量又指向一个整型变量。

指向指针的指针通常用来访问指针数组。数组可以保存基本数据类型的数据，同样，它也可以保存指针，保存指针的数组称为指针数组。如果要访问一个指针数组，使用指向指针的指针最为方便。例如：

```
// 这是一个指针数组，其中保存各个常量字符串的首地址
char* arrMonth[]={"Jan","Feb","Mar","Apr","May","Jun",
    "Jul","Aug","Sep","Oct","Nov","Dec"};
// 定义一个指向指针的指针
// 这里数组的首地址 arrMonth 就是数组的第一个元素，
// 也就是指向第一个字符串“Jan”的指针
char** pMonth = arrMonth;
// 获取用户输入
int nIndex;
cout<<"请输入月份对应的数字："<<endl;
cin>>nIndex;
// 对指针进行运算，使其指向相应的数组元素，
// 也就是相应的字符串指针
char* pCurMonth = *(pMonth + ( nIndex - 1 ));
cout<<"对应的月份是："<<pCurMonth<<endl;
```

在以上这段代码中，首先定义了一个指向字符型指针的指针 pMonth，并将字符串数组的首地址赋值给它，让其指向字符串数组的第一个元素。然后通过对 pMonth 进行运算，让其偏转指向数组中所对应的其他元素，这时“pMonth + (nIndex - 1)”就是指向这个字符串指针的指针。

7.1.4 指针在函数中的应用

指针因其在访问内存上的灵活性而出名，当它跟函数搭配使用的时候，可以大大提高函数的灵活性，增加函数的威力。指针在函数中的应用主要包括两个方面：指针作为函数参数

和指针作为函数的返回值。

1. 指针作为函数参数

在大多数情况下，函数之间参数的传递，都是通过传值来完成的。参数的传值传递要对数据进行拷贝，如果要传递的数据比较大，比如要向函数传递一个数组，则会增加函数调用的开销。这时，可以利用指针作为函数参数。在函数内部，利用指针能够灵活访问内存的特性，可以通过指针来访问需要传递进函数的数据，以此来完成大量数据的传递，减少函数调用的开销。使用指针作为函数参数，不仅可以向函数传入数据，还可以向函数的调用者传出数据。因为函数的调用者和函数都可以使用指向同一内存地址的指针，也就是使用同一块内存，所以使用指针作为函数参数时，就是对同一数据进行读/写操作，这样不仅可以传入数据，还可以通过在函数内部修改这些数据，把函数的结果传出给调用者。下面来看一个使用指针作为函数参数的实际例子。

```
// 计算数组中所有数据之和的函数
// 其中，参数 pArray 和 nArrayCount 分别表示数组的首地址和数组元素的个数，
// 用于向函数传入一个数组，
// nSum 表示最后求得的数组所有数据之和，用于从函数中传出计算结果
void SumArray(int* pArray, int nArrayCount, int* nSum )
{
    *nSum = 0;

    // 循环遍历整个数组
    for( int i = 0; i < nArrayCount; ++i )
    {
        // 通过指针访问数组元素，
        // 同时通过指针访问保存结果的变量，
        // 实际上也就是对主函数中 nArraySum 变量的访问
        // 将结果保存到 nArraySum 中，传出结果
        *nSum += *pArray;
        pArray++; // 指针加运算，访问数组中的下一个元素
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义保存结果的变量和数组
    int nArraySum;
    int nArray[5] = { 1, 2, 3, 4, 5 };

    // 使用数组的地址 nArray 传入数组，
    // 使用指向变量 nArraySum 的指针来接收计算结果
    SumArray(nArray, 5, &nArraySum);

    // 运算结果已经保存在 nArraySum 中，输出运算结果
    cout<<"数组中所有数据之和是 : "<<nArraySum<<endl;
```

```

    return 0;
}

```

在主函数中，调用了数组求和函数 SumArray()，并将数组的起始地址 nArray 和保存结果的整型变量的地址 &nArraySum 传递给了求和函数。在求和函数 SumArray() 中，通过传入的数组地址访问整个数组，完成传入数据的功能。同时，利用指向保存结果变量的指针，将结果直接保存到变量 nArraySum 中，完成传出数据的功能。利用指针作为函数参数传递数据的本质，就是在主调函数和被调函数中，通过指向同一内存地址的不同指针访问相同的内存区域，从而实现数据的传递和交换。图 7-3 展示了指针作为函数参数访问相同内存的结果。

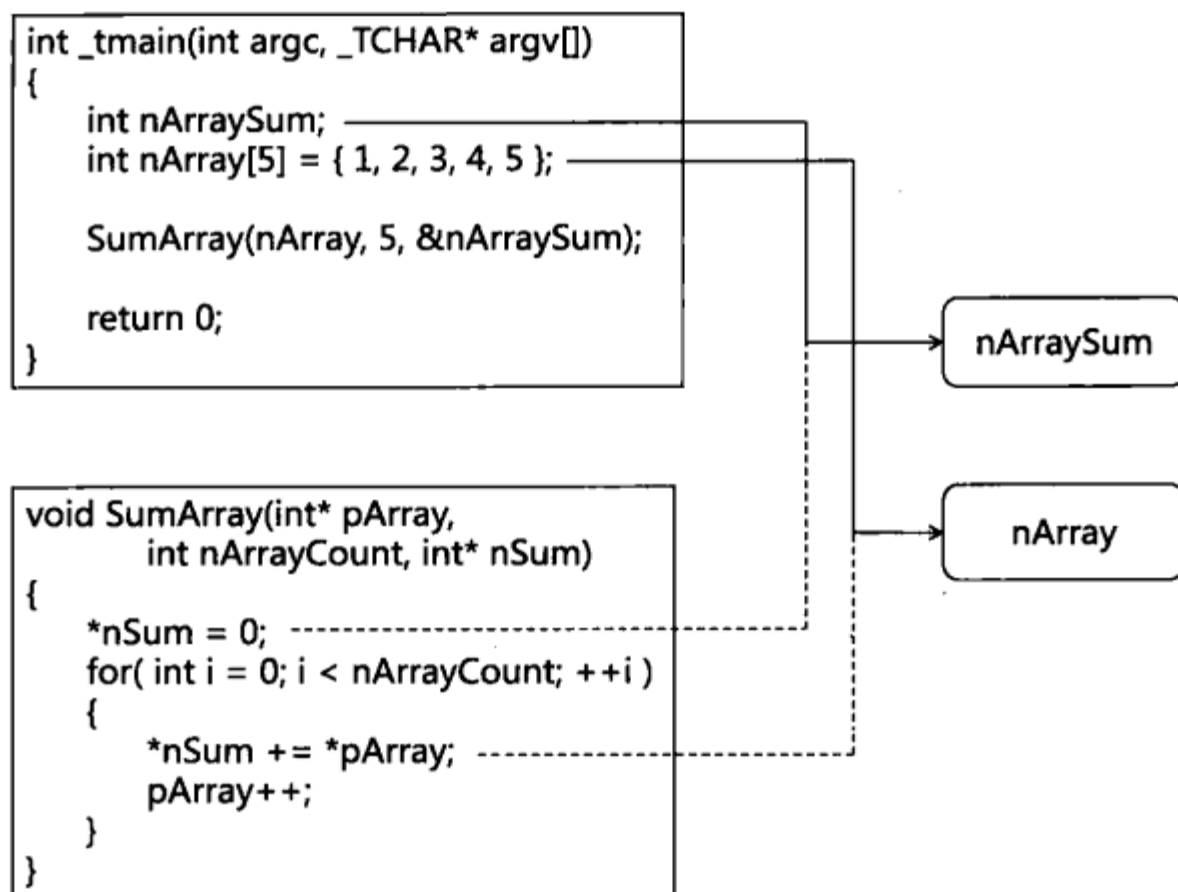


图 7-3 指针作为函数参数访问相同内存

2. 指针作为函数返回值

指针作为一种基本数据类型，同样可以作为函数的返回值。当函数的返回值是指针时，这个函数就是指针型函数。指针型函数通常用来获取一些指针变量的值，例如，在开发实践中经常使用的用于获得主窗口指针的函数 GetMainWnd() 就是一个指针型函数：

```

CWnd* GetMainWnd();

```

另外，使用指针作为函数返回值，还可以将大量的结果数据以指针的形式从被调函数返回到主调函数中，起到传出结果数据的作用。这点跟使用指针作为函数传出参数相似。

这里需要特别注意的是，不能把一个指向局部变量的指针作为返回值。这是因为函数内部声明的局部变量在函数结束后，其生命周期已经结束，内存会被自动释放，这时它的内存

地址是无意义的。如果将其作为函数返回值返回给主调函数，并在主调函数中访问这个指针所指向的数据，将产生不可预料的结果。

```
// 从函数中返回一个局部变量的指针
int* GetLocalVal()
{
    // 声明一个局部变量
    int nLocalVal = 1;
    // 返回局部变量的指针
    return &nLocalVal;
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 获取从函数中返回的局部变量指针
    int* pLocalVal = GetLocalVal();
    // 输出当前局部变量的值
    cout<<"当前局部变量的值是: "<<*pLocalVal<<endl;

    // 改变内存内容
    int a;

    // 再次输出局部变量的值
    cout<<"改变内存后, 当前局部变量的值是: "<<*pLocalVal<<endl;
    return 0;
}
```

执行程序后，可以得到这样的输出结果：

```
当前局部变量的值是：1。
改变内存后，当前局部变量的值是：1580854592。
```

从输出结果中可以清楚地看到，第一次输出时得到的是正确的结果。当声明一个新的整型变量、改变内存的内容再次输出时，得到的就是一个不确定的数值。这是为什么呢？因为调用的 `GetLocalVal()` 函数返回后，虽然局部变量 `nLocalVal` 已经不存在，但是存放它的内存中的值并没有改变，这时输出得到的还是它原来的值。但是当声明一个新的变量、改变这部分内存中的值时，则第二次输出的是一个不确定的值。因为在内存未改变时，通过指向函数局部变量指针得到的正确的值，往往使这种错误具有很大的隐藏性，这点尤其需要我们高度警惕，防止从函数中返回一个指向局部变量的指针。牢记下面的规则：指针函数可以返回全新申请的内存地址；可以返回全局变量的地址；可以返回静态变量的地址，但就是不可以返回局部变量的地址。

7.1.5 引用

在现实世界中，我们每个人基本上都会有好几个称呼：

长栓——妈妈叫的小名；

贾君鹏——户口本上的大名；

鹏程万里——网上自己取的昵称；

胖子——朋友给取的绰号。

虽然这些称呼各不相同，但都是指的同一个人。在 C++ 世界中，也有这样的现象，一个变量可能有多个名字，虽然名字不同，但都是指的同一个人，通过这些名字，我们都可以访问到这个变量。这种变量的别名在 C++ 世界中有一种更专业的称呼：“引用”。

引用的本质，就是变量的别名，通俗地讲，就是变量的绰号。对变量的引用的任何操作，就是对变量本身的操作，就像不管是叫你的小名，还是叫你的绰号，都是在叫同一个人。

声明引用的语法格式如下：

数据类型 & 引用名 = 变量名；

其中，数据类型跟要引用的变量的数据类型相同；其后的“&”符号表示声明的是一个引用，引用名跟变量名相似，也就是某个数据的第二个名字。例如，声明一个整型变量的引用：

```
// 首先定义一个整型变量
int nIntValue = 1;
// 定义一个整型引用并将它跟整型变量关联起来
int& rIntValue = nIntValue;
```

这样，就声明了 rIntValue 是变量 nIntValue 的引用。因为引用只是某个变量的别名，所以任何对变量引用的操作都相当于操作变量本身。例如：

```
int nIntValue; // 声明 int 变量
// 声明 int 引用，并跟变量 nIntValue 建立引用关系
int& rIntValue = nIntValue;

// 通过变量直接修改变量的值
nIntValue = 1;
cout<<"通过变量直接修改后，"<<endl;
cout<<"变量的值为"<<nIntValue<<endl;
cout<<"引用的值为"<<rIntValue<<endl;

// 通过引用修改变量的值
rIntValue = 2;
cout<<"通过引用间接修改后，"<<endl;
cout<<"变量的值为"<<nIntValue<<endl;
cout<<"引用的值为"<<rIntValue<<endl;
```

程序运行后可以得到以下这样的输出结果。

通过变量直接修改后，
变量的值为 1
引用的值为 1
通过引用间接修改后，
变量的值为 2
引用的值为 2

从输出结果中可以看到，无论是直接修改变量 `nIntValue` 还是间接修改引用 `rIntValue`，都是修改变量 `nIntValue`，从而验证了对一个变量的引用的修改，就是对这个变量本身的修改。

注意，引用在声明的时候必须初始化，否则会产生一个编译错误。就像给一个人取绰号，只有人存在了，我们才能给他取绰号。如果连人都不存在，那么绰号何从谈起？

大家可能已经注意到，前面介绍的指针跟引用有非常相似的功能：它们就像一对孪生兄弟，都是某个变量数据的指代，都可以以一种间接的方式访问它们所指代的变量。虽然是孪生兄弟，但肯定还是有细微的差别，那么，指针和引用的差别又在哪里呢？从使用规则上，指针和引用就有很大的不同。

- 初始化的要求不同。引用在声明的时候必须初始化，而指针则可以在任何合适的时候完成初始化。例如：

```
int x = 0;
int* pInt;           // 指针在声明时可以不进行初始化
pInt = &x;           // 在需要的时候完成指针的初始化
int& rInt = x;       // 引用在声明的时候必须初始化，rInt 是变量 x 的引用
```

- 跟变量关联的紧密性不同。引用只是变量的别名，不可能存在空的引用，也就是说引用必须与某个合法的事先存在的变量关联，而指针则可以为空指针（NULL），不与任何变量建立关联。例如：

```
int* pInt = 0;       // 一个空指针
// 声明一个引用，并跟这个空指针所指向的数据建立关联
// 这时 rInt 引用的是内存地址为 0 的整型数，虽然这个数并没有意义
int& rInt = *pInt;
```

- 对重新关联的要求不同。引用一旦被初始化，成为某个变量的别名，就不能改变这种引用关系跟其他的变量关联。引用跟它所关联的变量之间的关系是从一而终、固定不变的，而指针则可以随时改变所指向的变量，有点水性杨花，说变就变。例如：

```
// 定义另外一个整型变量
int y = 1;
// 这条语句不是重新改变 rInt 引用的变量，
// 而是对其进行赋值，此时引用 rInt 和变量 x 的值都是 1
rInt = y;
// 重新改变指针 pInt 所指向的变量，此时指针 pInt 的值发生了变化
pInt = &y;
```

取绰号的目的是什么？没错，是为了别人称呼起来更加方便。引用的意义也是为了让它所关联的变量使用起来更加方便。在进行普通运算的时候，大都是变量自己亲自出马，无须引用这个替身出场，但是到了变量作为函数参数，或者作为函数返回值在函数之间进行频繁传递的时候，就该引用上场了。引用的主要应用就是传递函数的参数和返回值。例如：

```
// 给整型数加 1
void Increase( int& nVal )
{
    nVal += 1;
}

int nInt = 1;
Increase( nInt ); // 变量 nInt 的值变为 2
```

这里利用了引用作为 Increase()函数的参数。当用一个整型变量调用 Increase()函数时，实际上是用这个整型变量对引用参数进行初始化。在函数内部，引用所关联的变量就是调用函数时所使用的变量，所以，在函数内部对引用的访问也就相当于访问变量本身，这样同时实现了函数数据的传入和传出。

现在回顾前面的学习，已经知道以下三种传递函数参数和返回值的方法。

- 传值。传值是指直接将实际参数的值复制给形式参数，完成参数的传递。
- 传指针。传指针是指将需要传递的数据的指针作为参数进行传递。
- 传引用。传引用是指将需要传递的数据的引用作为参数进行传递。

人没有选择，很苦恼，但是选择太多，会更加苦恼。传递函数参数和返回值的形式这么多，它们有何差别？我们又该如何选择？不用迷惑，下面列举一个例子来比较这三种传递参数的方式，就知道该怎么选择了。

```
// 通过传值来传入参数和传出返回值
int FuncByValue(int x)
{
    x = x + 1;
    return x;
}

// 通过传指针来传入参数和传出返回值
void FuncByPointer(int* p)
{
    *p = *p + 1;
}

// 通过传引用来传入参数和传出返回值
void FuncByReference(int& r)
{
    r = r + 1;
}
```

```

int _tmain(int argc, _TCHAR* argv[])
{
    int n = 0;
    cout<<"n 的初始值, n = "<<n<<endl;

    // 调用传值方式的函数, 变量 n 的值不发生改变
    FuncByValue( n );
    cout<<"传值, n = "<<n<<endl;

    // 调用传指针的函数, 实现数据的同时传入传出
    // 变量 n 的值发生改变
    FuncByPointer( &n );
    cout<<"传指针, n = "<<n<<endl;

    // 调用传引用的函数, 实现数据的同时传入传出
    // 变量 n 的值发生改变
    FuncByReference( n );
    cout<<"传引用, n = "<<n<<endl;

    return 0;
}

```

运行程序后可以得到这样的输出结果：

```

n 的初始值, n = 0
传值, n = 0
传指针, n = 1
传引用, n = 2

```

从程序的输出结果可以分析这三种传递参数方式的区别：在函数 `FuncByValue()` 中，其内部的形式参数 `x` 只是外部实际参数 `n` 的一份拷贝，当对 `x` 进行运算时不能改变 `n` 的值，所以输出结果仍然为 0；在函数 `FuncByPointer()` 中，指针 `p` 是指向外部变量 `n` 的指针，改变指针 `p` 所指向的数据的值实际上就是改变 `n` 的值，所以在函数内部对指针 `p` 所指向的数据的改变，就直接反映到 `n` 的数值的修改，输出结果为 1；在函数 `FuncByReference()` 中，引用 `r` 就是外部变量 `n` 的引用，引用 `r` 和变量 `n` 都是同一个数据，在函数内部改变 `r` 同样也会修改 `n`，所以最终输出结果为 2。对比以上三种传递参数的方式可以发现：传引用的性质跟传指针相似，既可以传入参数也可以传出参数；同时传引用又跟传值的书写形式相似，它可以直接使用变量调用函数，在函数内部引用的使用方式也跟普通变量的使用方式一样。这样传引用既享受了传指针的好处——节省空间，提高效率，可以用做参数的传入和传出，又跟传值保持了相同的书写形式——让引用在函数中使用起来更加简单自然，代码的可读性也更高。可以说，传引用同时具备了传指针和传引用两者的优点。所以，在可以的情况下，都应该尽量使用传引用来传递函数参数，而尽量少用传指针（指针使用具有一定的危险性）和传值（效率低下）这两种形式。简单来讲，就是尽可能地使用引用，不得已时才使用指针。三种传递参数的方式如图 7-4 所示。

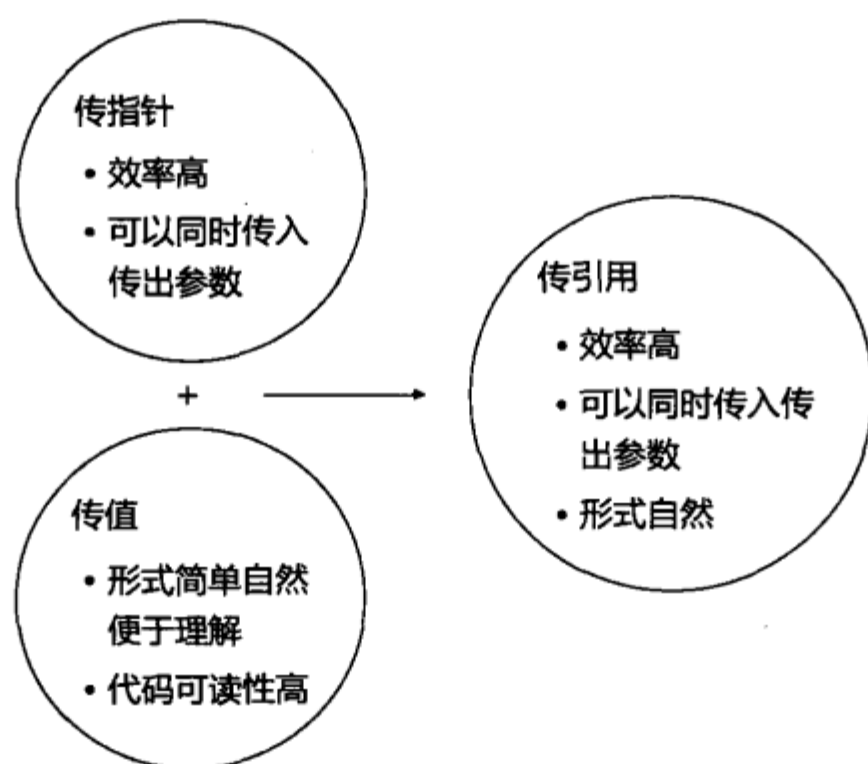


图 7-4 优先选择传引用

7.2 程序中的异常处理

“这是哪个笨蛋程序员写的程序，程序出错又崩溃了！”

“这个程序一打开文件就死了，怎么回事啊？”

“就因为程序的崩溃，我一上午的心血啊，如同滔滔江水滚滚东去了！”

我们是不是经常听到这样咒骂程序和程序员的声音？这些都是因为应用程序没有进行很好的异常处理，程序运行遇到错误异常时导致程序无法响应，甚至造成系统崩溃。程序员也是人，是人就会犯错误。同时程序运行的情况千差万别，所以程序在运行过程中出现错误也在所难免。作为一个程序员，程序出现异常不是你的错，但是出现异常后不进行异常处理就是你的错了。所以，如果不想被人骂作笨蛋，如果不想让用户的成果付诸东流，现在就来学习 C++ 中的亡羊补牢之术——异常处理吧！

7.2.1 异常处理

在程序运行过程中，因为运行情况的不同可能会产生一些错误，例如，尝试打开某个已经被占用的文件而导致文件打开失败无法写入数据、数组下标访问越界、系统内存不足等。一旦出现这些问题，则可能引发算法失效、程序运行时无故停止，甚至程序崩溃等。因此也就出现了本节开始的那一幕幕惨剧，所以这就要求在开发程序的时候进行必要的异常处理，亡羊补牢。比如，针对文件打开失败的情况，处理的方法有很多，最简单的就是使用“return”

语句，告诉上层调用者函数执行失败，不再进行数据的写入等后继操作；另外一种处理策略就是利用 C++ 的异常机制，抛出异常并对异常进行恰当处理。

C++ 异常处理机制给程序提供了一种对在运行时出现的意外或异常情况进行处理的方法。异常处理使用 try 关键字来尝试执行可能会出现异常的代码段，当在代码段的执行过程中有异常抛出时，由 catch 和 finally 关键字表示的异常处理语句会对异常进行恰当的处理，比如结束正在执行的操作、事后清理资源等。在 C++ 世界中，异常处理的语法格式如下：

```
// 用 try 开始异常处理语句
try
{
    // 包含可能抛出异常的语句
}
catch(异常类型名 [形参名])    // 捕获特定类型的异常
{
    // 对异常进行处理
}
catch(异常类型名 [形参名])    // 捕获特定类型的异常
{
    // 对异常进行处理
}
// 可以有多个 catch 语句并列，捕获不同的异常
catch(...)    // 如果省略具体的异常类型用“...”表示，则表示捕获所有类型的异常
{
    // 对所有类型的异常进行处理
}
finally:
{
    // 对异常的最终处理，对所有异常的处理都会执行这些语句
}
```

在这段代码中，使用 try 关键字开始处理语句，其中包含可能抛出异常的语句。当程序执行 try 语句块中的语句（包括其中调用的函数）时，如果遇到程序执行发生错误，就使用 throw 关键字抛出一个异常。throw 关键字的语法格式如下：

throw 异常表达式；

其中，异常表达式就是要抛出的异常，它可以是某个具有某种意义的错误代码，或者是含有异常相关信息的数据，总之，它的意义是为异常处理提供相应的辅助信息。例如：

```
// 除法函数
double Divide( int a, int b )
{
    if( 0 == b )
        throw "不能使用 0 作为除数";

    return (double)a/b;
}
```

在这个除法函数中，当检测到除数为 0 时，它就用 throw 关键字抛出一个异常提前结束这个函数，跳过后面的除法运算。这里 throw 关键字抛出的异常是一个字符串，它描述了这个错误相关的信息。当然，还可以抛出其他类型的异常，只要这个异常是事先定义好的并能够给后面的异常处理提供足够信息就可。

当 try 语句块抛出异常时，该异常会被紧跟其后的 catch 语句捕获并进行相应的异常处理。catch 语句可以带有一个形式参数，它的类型就是 catch 语句要捕获的异常类型，也就是说，异常被某个 catch 语句捕获并且处理的条件就是该异常的类型与 catch 语句的异常类型相匹配。当使用 throw 关键字抛出的异常被某个 catch 语句捕获时，throw 关键字后的异常表达式会被当成参数传递给 catch 语句，catch 语句则可以根据这个参数对异常进行具体的处理。例如，如果使用 throw 关键字抛出的是一个字符串类型的异常，它就会被参数是字符串类型的 catch 语句捕获，然后输出这个字符串并告诉用户发生了异常。因此，判断异常将被哪一个 catch 语句处理时，throw 语句中的表达式的值并没有太多的实际意义，而表达式的类型却特别重要。

当有多种类型的异常需要捕获时，可以将多个 catch 语句并列。如果省略 catch 关键字后面的形式参数而使用“...”代替，就表示 catch 语句会捕获所有类型的异常。在完成对所有具体异常的处理之后，还需要对所有异常进行最终的处理，比如写入日志、重启应用程序等。所有这些异常处理要执行的动作都可以放在 finally 语句中来完成，因为无论 try 语句块抛出的是什么类型的异常，在执行完具体的异常处理 catch 语句之后，finally 语句总会被执行。

当 try 语句块发生错误时，使用 throw 关键字抛出相应的异常，比如错误的内存申请会抛出 std::bad_alloc 类型的异常。异常处理会把控制权从异常发生的地点转移到与这个异常相匹配的 catch 语句，开始对异常进行处理。例如：

```
// 开始异常处理语句
try
{
    // 捕获 Divide() 可能会抛出的异常
    // 这里使用 0 作为除数，将抛出一个异常
    double fResult = Divide( 3, 0 );
}
// 捕获 try 语句块中所抛出的字符串异常
catch( char* pMsg )
{
    // 对异常进行处理
    // 这里仅仅是输出错误信息
    cout<<"程序运行发生异常: "<<pMsg<<endl;
}
```

在这段代码中，在 try 语句块中调用了 Divide() 函数，而 Divide() 函数在执行过程中会对除数进行检测，当检测到除数为 0 时，它就用 throw 关键字抛出一个描述了错误信息的字符串异常。在 try 语句块之后的 catch 语句会捕获到这个字符串异常并对其进行处理，这里对异

常的处理只是将这个错误信息输出报告给用户。在实际应用中，异常处理往往要比这复杂，比如要进行资源的清理、记录错误日志等。异常处理的三个步骤如图 7-5 所示。

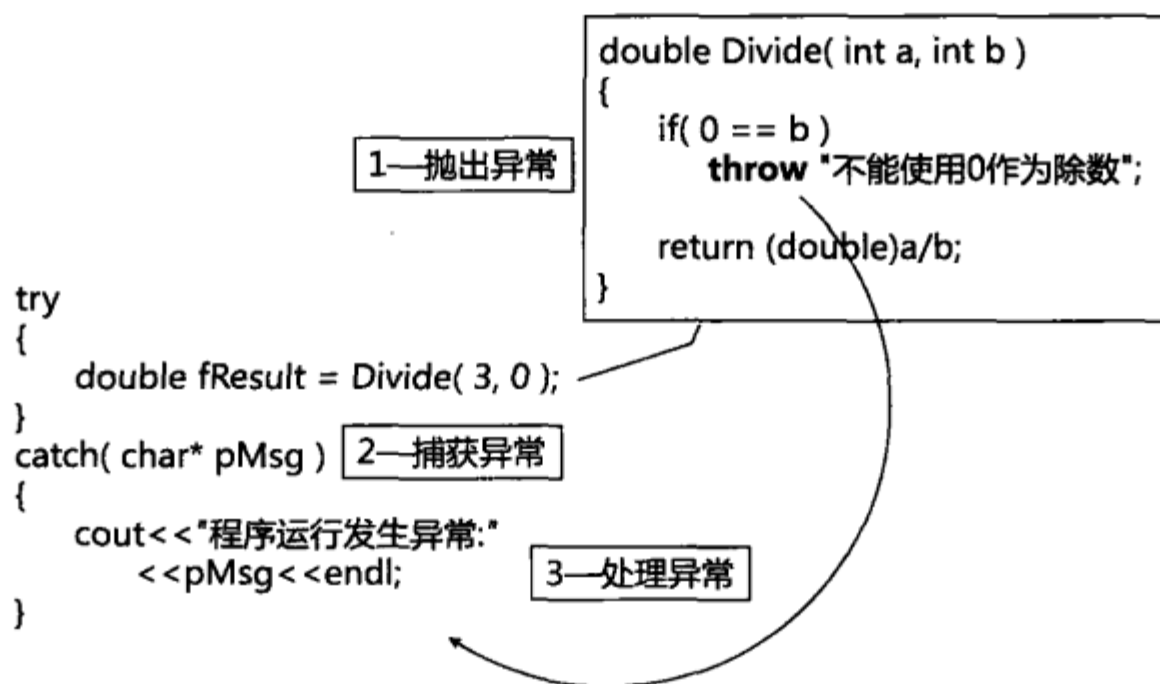


图 7-5 异常处理三部曲

总之，使用异常处理这“亡羊补牢”之术，可以在错误发生之后对错误所造成的损失尽量进行补救，让用户的损失减到最小，也让程序更加健壮。

7.2.2 异常的函数接口声明

为了加强程序的可读性，方便函数使用者知道所使用的函数可能会抛出的异常类型，可以在函数的声明中列出这个函数可能抛出的所有异常类型，从而进行有针对性的异常处理。函数的异常接口声明格式如下：

返回值类型 函数名(形式参数表) throw (异常类型列表);

例如，Divide()函数可能会抛出字符串类型和浮点数类型，可以将函数声明如下：

```
double Divide( int a, int b ) throw ( char*, double )
{
    // ...
}
```

这里，使用函数名之后的 throw 关键字限制了 Divide()函数所能够抛出的异常类型只能是字符型指针类型和浮点数类型。如果尝试在 Divide()函数中抛出其他类型的异常，将产生一个编译警告。当函数运行时，如果真的抛出函数接口声明中未列出的异常，就会导致异常冲突。因为没有提示函数的调用者：该函数会抛出一种没有被说明的即不期望的异常，于是异常处理机制就会检测到这个冲突并调用标准库函数 unexcepted()来处理这个异常，unexcepted()函数的默认行为就是调用 terminate()函数来结束程序，最终导致该异常得不到正确的处理。所

以，根据函数接口声明抛出正确类型的异常对异常得到正确的处理很重要。

如果在函数的声明中没有包括异常的接口声明，则此函数可以抛出任何类型的异常。例如：

```
double Divide( int a, int b );
```

相反，如果将 throw 关键字之后的异常类型列表留空，则表示这个函数不会抛出任何类型的异常。例如：

```
double Divide( int a, int b ) throw();
```

合理使用异常的函数接口声明，可以让函数在抛出异常方面更加清晰明确，在调用函数并对其中可能抛出的异常进行处理时也更具针对性。

7.2.3 合理使用异常处理

既然异常处理可以防止程序出错，那么是不是在程序的任何地方都需要使用异常处理呢？当然不是！虽然异常处理可以很好地对程序中可能发生的异常进行处理，但是如果使用不当，它也会带来性能上的损失。所以只在合适的地方使用异常处理，防止异常处理的滥用。

1. 不要用异常替代可以处理的分支结构

在程序中，往往有可以预期的分支结构，在各分支结构中可针对不同的情况进行处理。我们可以用返回值来表示一个分支状态，而不是抛出一个异常；我们不能用异常来完全代替返回值，因为返回值的含义不一定只是成功或失败，有时候是一个可选择的状态。例如：

```
bool bResult = IsFinished();
if( true == bResult )
{
    return true;
}
else
{
    return false;    // 不要用抛出异常来代替可以明确表示状态的返回值
}
```

在这种情况下，不论返回值是什么，都是程序可以接受的正常结果。而异常只能用来代表“异常”，也就是非预期的错误状态。

2. 将异常用在发挥它优点的地方

异常的使用，会降低程序的性能，但是，如果使用得当，有时也可以提高程序的性能。比如，如果函数的参数是指针，则需要在函数入口处对这个指针的有效性进行检查。

```
bool Check( int* pPointer )
{
    if( NULL == pPointer )
```

```
        return false;

    // 指针有效，可以访问指针
}
```

如果使用异常处理，就可以避免对指针的有效性进行检查而直接访问，只需要在指针无效、访问发生异常时对抛出的异常进行处理就可：

```
bool Check( int* pPointer )
{
    try
    {
        // 直接访问指针
    }
    catch(...)
    {
        // 对指针无效的异常情况进行处理
        return false;
    }
}
```

如果指针参数为 NULL 的几率非常小，那么每次函数调用都对指针参数进行检查会造成浪费，这时可以直接使用这个指针，并且仅在指针为 NULL 的时候进行异常处理。因为指针参数为 NULL 的几率非常小，所以异常处理发生的几率也非常小，最终整个程序的效率更高。

3. 尽量不要在循环中使用异常处理

因为异常处理需要大量的额外操作，所以它并不适用于经常运行的代码，比如循环中。如果要对循环中发生的错误进行处理，使用返回值是一个好的选择。

7.3 编写更复杂的 C++ 程序

见过 C++ 世界的指针和异常处理这两位“高人”之后，大家是不是感觉自己的 C++ 功力倍增，开始有点高手的感觉了？

现在，我们已经能够利用学到的知识编写简单的 C++ 程序了。但是，高手可不是仅仅能够写简单的 C++ 程序就够的。要想成为一个真正的高手，还要继续向 C++ 世界的更高峰攀登，学习如何编写更加复杂的 C++ 程序。

7.3.1 源文件和头文件

通过前面的学习，我们是否注意到编写的程序代码已经越来越长：从最开始的几行代码到几百行代码。而在实际开发中，一个软件往往会有上十万行、百万行甚至千万行的代码。面对这么多行代码，不可能把它们都写在同一个源文件中，为了便于理解和管理，往往根据

需要将其中不同的逻辑实现放在不同的源文件中，将相关的声明放在对应的头文件中。一个程序的所有代码都包含在它的源文件或头文件中。

1. 源文件

源文件以“cpp”等为文件后缀名，它主要用于实现程序的各种功能。比如，可以将程序划分为多个子模块，每个模块单独放在一个源文件中进行管理。

2. 头文件

头文件以“h”等为文件后缀名，主要包含函数、数据（包括数据类型的定义）、类等等的声明。因为头文件可以被多个源文件引用，所以可以在多个源文件内共享这些函数和数据的声明，以达到共用的目的。比如，在前面的例子中多次引用“iostream”这个头文件，就是为了共享声明在其中的输入/输出流对象。

大多数时候，源文件和头文件是成对出现的，我们把某个子模块的声明放在头文件中，而将其具体的实现放在对应的源文件中。比如，常常把一个类的声明放在头文件中，而把它的具体实现放在源文件中。在一个程序的所有源代码文件中，源文件和头文件各司其职，共同完成一个完整的程序。源文件和头文件不仅在内容上有差别，在使用上也有很大的差别。

3. 从引用的角度讲

源文件可以通过引用一个或多个头文件，达到共用其中各种声明的目的，但是头文件不可以引用源文件。虽然头文件引用源文件在语法上是允许的，但是在实际使用中是不规范的做法。另外，头文件还可以引用其他的头文件，从而构成新的头文件。比如在项目中经常见到的 `stdafx.h`，实际上就是多个头文件的集合。

```
// stdafx.h : 标准头文件

#pragma once

// 引入项目所需求的其他标准头文件
#include "targetver.h"

#include <stdio.h>
#include <tchar.h>

// TODO: 添加其他需要的头文件
```

这样，在源文件中引用 `stdafx.h` 这个头文件，就相当于间接引用了 `targetver.h` 和 `stdio.h` 等 `stdafx.h` 所引用的头文件。

4. 从预处理的角度讲

对于包含了多个头文件的源文件，只要用头文件的内容替换掉源文件中对应的 `#include` 语句就可以得到预处理后的源文件。这种经过预处理后生成的源文件的编译结果就是最终的

编译结果。换句话说，头文件可以看成是多个源文件的共有部分。这些共有部分被抽取出来成为一个头文件。反过来，这个头文件又被多个源文件引用，以此来达到代码复用的目的。源文件和头文件的包含关系如图 7-6 所示。

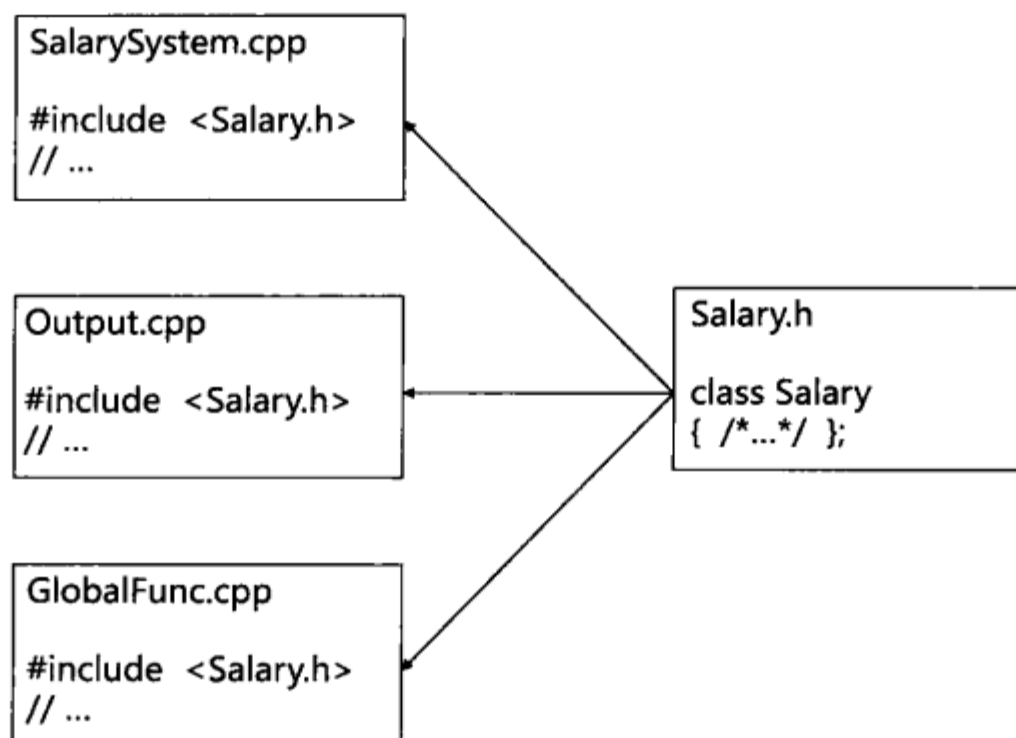


图 7-6 源文件和头文件的包含关系

5. 从编译的角度讲

程序的源文件会被直接编译，也就是说，源文件会被直接编译为.obj 文件。而头文件如果没有被任何源文件引用，就不会被编译。换句话说，如果头文件没有被任何源文件直接或者间接引用，那么这个头文件就相当于一个多余的无用文件，不会参与整个程序的编译过程，即使其中有语法错误也不会在编译时检测出来。另外，因为一个头文件可能被多个源文件引用，为了防止头文件被多次编译，导致其中所声明的变量或者函数被重复多次声明，所以需要一些预编译指令来防止一个头文件被多次编译。通常，可在头文件中加入“#pragma once”预编译指令来防止一个头文件被多次编译。例如：

```
// stdafx.h : 标准头文件

#pragma once
// 声明语句或者引入其他头文件
```

当编译器看到“#pragma once”预编译指令之后，将由它来保证这个头文件只被编译一次。

7.3.2 名字空间

在一些大型系统中，因为多人参与共同开发，所以即使采用了源文件、头文件分离的方式，也难以保证函数、数据声明的唯一性。比如，程序员张三声明了一个名为 Student 的数据

类型，而李四在不知情的情况下，同样声明了另外一个名为 Student 的数据类型，这样在同一个系统中的两个不同的 Student 就会产生冲突。为了解决这个问题，C++ 世界提供了名字空间（namespace）来规划和管理程序结构。

如果说函数是一个箱子，那么名字空间就像一个仓库。把函数或者变量的声明和定义存放在某个仓库中，即使在其他仓库中有相同名字的函数或者变量，两者也可以互不影响。也就是说，不同名字空间内的同名函数、同名变量等可以同时存在，互不冲突。比如，张三把 Student 声明在名字空间 Zhangsan 中，而李四把 Student 声明在名字空间 Lisi 中，这样就不会冲突了。定义一个名字空间的语法格式如下：

```
namespace 名字空间名
{
    // 名字空间内的声明和定义
};
```

其中，用 namespace 关键字表示名字空间的开始，其后的名字空间名就是这个名字空间的名字，它通常是一个可以用来表示这个名字空间的意义的易于识别的名词。定义好名字空间之后，就可以在其中进行其他各种声明和定义了，在名字空间中声明和定义的内容都属于这个名字空间。

现在，就可以用名字空间来解决上面的结构体冲突问题了，例如：

```
// 名字空间 Zhangsan
namespace Zhangsan
{
    // 名字空间 Zhangsan 中的 Student 数据结构
    struct Student
    {
        int nIndex;
        int nAge;           // 年龄
    };
};
// 名字空间 Lisi
namespace Lisi
{
    // 名字空间 Lisi 中的 Student 数据结构
    struct Student
    {
        int nIndex;
        string strName;    // 姓名
    };
};
// 全局名字空间
// 如果没有说明在哪个具体的名字空间，则默认在全局名字空间
struct Student // 全局名字空间中的 Student 数据结构
{
    int nIndex;
```

```

        bool bMale;           // 性别
    };

```

从以上代码中可以看到，在名字空间 Zhangsan、名字空间 Lisi 和全局名字空间中分别定义了 Student 结构体，并且 Student 结构体的实现各不相同，如图 7-7 所示。这是因为 Student 结构体分别属于不同的名字空间，所以，虽然这些结构体的名字相同，但是并不会产生冲突，就像你家有电视机，我家也有电视机，但实际上却是两个不同的电视机，互不影响。

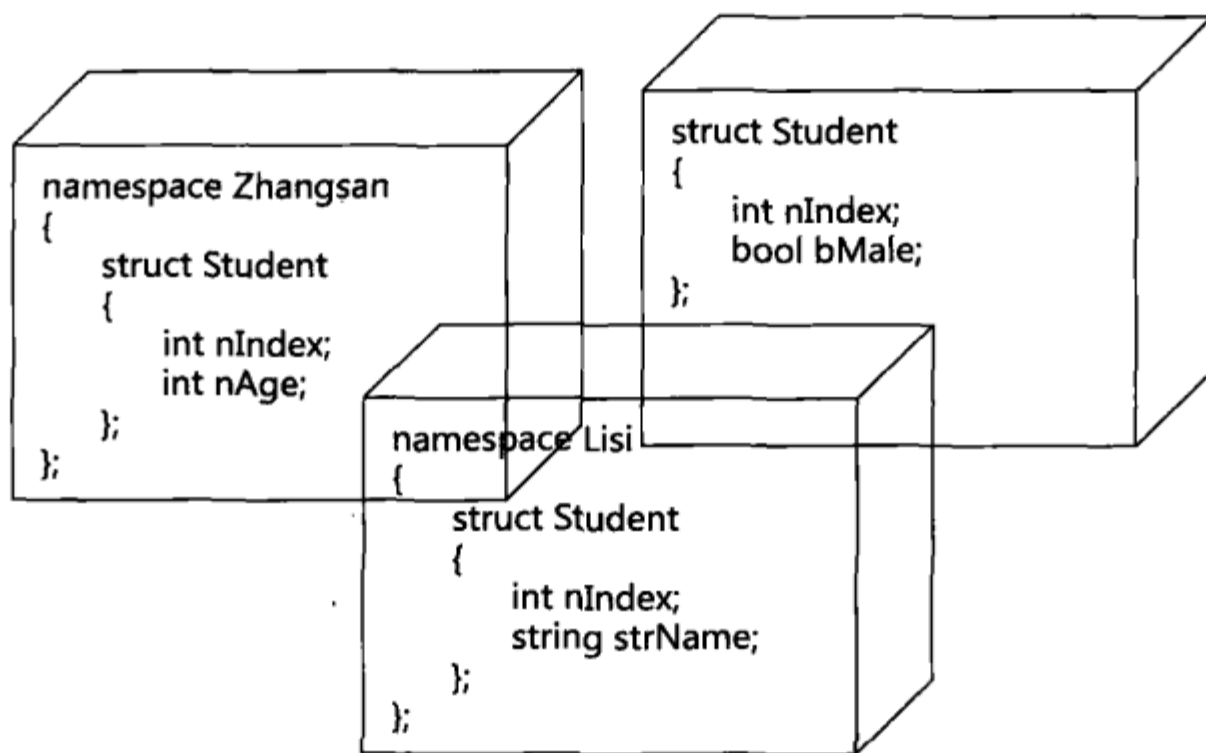


图 7-7 名字空间的包装功能

这么多同名的结构体，又如何区分呢？正如我们说“这是贾玮家的电视机”一样，可以在这些同名的结构体前加上相应的名字空间来限制修饰结构体，从而访问在某个名字空间下所定义的具体的某个结构体。例如：

```

// 声明一个类型为 Zhangsan::Student 的变量
// 这时使用的是 Zhangsan 名字空间下的 Student 结构体
Zhangsan::Student    zStudent;
zStudent.nAge = 14;
// 声明一个类型为 Lisi::Student 的变量
// 这时使用的是 Lisi 名字空间下的 Student 结构体
Lisi::Student    lStudent;
lStudent.strName = "Chen Liangqiao";
// 声明一个类型为 ::Student 的变量，
// 如果结构体前没有指明名字空间，则默认为全局名字空间
// 结构体::Student 等同于 Student
::Student    gStudent;
gStudent.bMale = true;

```

从这段代码中可以看到，在使用名字空间时，可以使用“::”域操作符来说明其后的数据类型是哪个名字空间下的数据类型。比如，Zhangsan::Student 就表示所使用的是名字空间 Zhangsan

下定义的数据结构体 Student，它包含的是一个索引值和学生的年龄。特别地，如果“::”没有显式地给出一个空间名字，就说这是全局名字空间，也可以称为匿名名字空间。任何没有在给定名字空间下的声明都是匿名名字空间下的，比如上面例子中的第三个 Student 结构体。

在实际开发中，为了简化代码，并不需要每次都显示地指明函数、结构体等所在的名字空间，通常用“using namespace”关键字来指明编译时默认查找的名字空间。例如，如果在源文件中添加了“using namespace Zhangsan”语句，就表示将 Zhangsan 这个名字空间同样作为默认查找的名字空间。当编译器遇到 Student 这个数据类型时，就会在各个默认的名字空间中查找相应的声明和定义。如果只有一个名字空间内有这样的类型，编译器就会自动给它加上名字空间的前缀；如果编译器发现某个名字在多个默认的名字空间同时存在，就会出现编译错误，这时必须使用“::”域操作符显式地指定类型所在的名字空间了。例如：

```
// 名字空间 Zhangsan
namespace Zhangsan
{
    struct Student
    {
        // ...
    };

    struct Teacher
    {
        // ...
    };
};

// 名字空间 Lisi
namespace Lisi
{
    struct Student
    {
        // ...
    };
};

// 全局名字空间
struct Student
{
    // ...
};

// 引入名字空间 Zhangsan
// 同时默认使用全局名字空间
using namespace Zhangsan;

int _tmain(int argc, _TCHAR* argv[])
{
    // error C2872: "Student": 不明确的符号
    // 因为默认名字空间 Zhangsan 和全局名字空间内
```

```

// 都有 Student 的定义，编译器无法确定到底使用哪个定义。
// 这里应该使用::Student 表示全局名字空间中的定义，
// 或者使用 Zhangsan::Student 表示 Zhangsan 名字空间中的定义
Student zStudent;

// 明确的类型，因为只有名字空间 Zhangsan 中有 Teacher 的定义
Teacher zTeacher;
// 明确的类型，显式地指定了名字空间为 Lisi
Lisi::Student lStudent;

return 0;
}

```

在这段代码中，使用 using 关键字引入了名字空间 Zhangsan 作为默认的名字空间。当编译器查找 Student 的定义时，会发现在全局名字空间和名字空间 Zhangsan 下都有 Student 的定义，这时就会产生编译错误，使用默认的名字空间就无法区分这个结构体，必须显式地使用域操作符“::”指明它所在的名字空间。而对于 Teacher 在默认名字空间（包含全局名字空间和名字空间 Zhangsan）只有唯一定义的结构体，则可以省略结构体前名字空间的修饰，直接使用结构体名进行变量的定义。当然，如果要明确地使用某个名字空间下的定义，则可以直接显式地指明这个定义所在的结构体，只要不怕麻烦。

在开发实践中，常常将不同的模块划分为不同的名字空间，这样各个模块内部声明的函数、数据等不会互相影响，名字空间起到了很好的模块包装的作用。

7.3.3 作用域与可见性

随着开发程序的功能越来越多，程序规模也越来越大，其中涉及对共享数据或者函数的管理也变得更加复杂。就像仓库中存放的东西越来越多，就有必要对这些东西进行管理一样。虽然可以使用箱子将程序的各个子功能封装成函数，但是当函数越来越多时，还是要采用分而治之的原则，利用更高抽象程度的区域划分将多个函数或者变量划分到不同的区域进行管理，这个划分的区域就是 C++ 世界中标识符的作用域。

作用域定义了某个标识符在程序中有效的区域。可见性是从另外一个角度——标识符引用来查看作用域的。如果在某个作用域内标识符是有效的、可以引用的，就说这个标识符在这个作用域内可见。换句话说，就是标识符只在其作用域内可见。

在 C++ 程序中，按照作用区域的大小，可以把作用域分为局部作用域和全局作用域。

1. 局部作用域

在 C++ 程序中，用大括号（{}）括起来的代码范围属于一个局部作用域。作用域可以嵌套作用域，如果局部作用域中包含更小的子作用域，那么子作用域具有较高的优先级，也就是说，在父作用域中可见的标识符在子作用域中同样可见。在局部作用域内，一个变量或者

函数从其声明的位置开始，一直作用到该作用域结束为止。常见的局部作用域有函数体，以及 if、for、while 和 switch 等复合语句。例如：

```
int GetSum( void )
{
    // 整个函数体是一个局部作用域
    int nTotal = 0;
    for( int i = 0; i < 100; ++i )
    {
        // 函数体中的 for 循环体，是函数体所嵌套的一个子局部作用域
        // 在父作用域函数体中定义的变量，在子作用域中同样可见
        nTotal += i;
    }

    // 在作用域中定义的变量，在整个作用域都可见
    return nTotal;
}
```

当然，为了更好地管理程序中的函数或者变量，也可以根据需要人为地在代码中添加一对“{}”来构成一个局部作用域。例如：

```
void foo(void)
{
    int nNum = 0;
    {
        int nNum;
        nNum = 1;
        cout<<"在局部作用域中输出："<<nNum<<endl;
    }
    cout<<"在函数体作用域中输出："<<nNum<<endl;
}
```

在这个例子中，在函数体作用域的开始定义了变量 nNum，因此它的作用域就是其定义后从函数开始到函数最后结束。第二个变量 nNum 定义在一个局部作用域中，因此它的作用域是其定义后从开始到“{}”语句块最后结束。虽然是相同的变量名，但是，因为处在不同的作用域中，所以不会产生冲突；又因为子作用域的优先级高于父作用域，如“nNum = 1”赋值语句实际上是对第二个局部作用域中的变量 nNum 进行赋值，不会影响父作用域中的 nNum 变量，所以最后输出函数作用域内的变量 nNum 的值还是最开始的初始值。虽然这样的做法在 C++ 语法上是允许的，但是为了避免代码在语义上的混淆，最好不要采用这样的方式在一个函数中定义两个同名的变量，如图 7-8

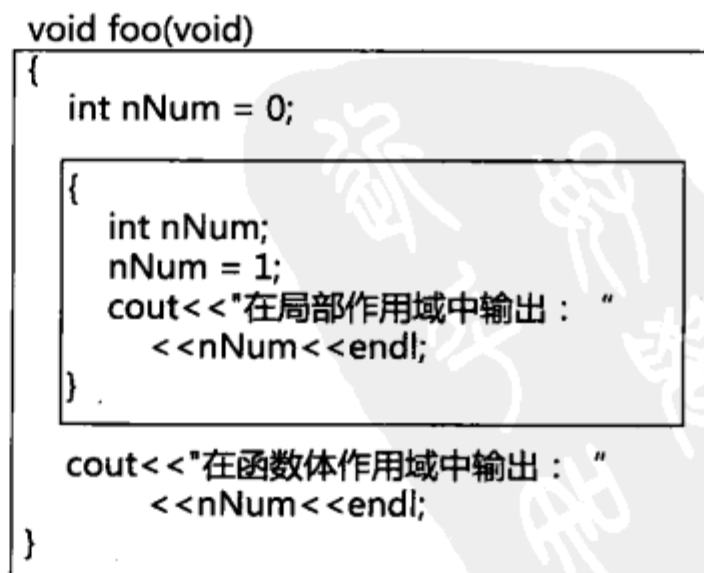


图 7-8 作用域与局部作用域

所示。

2. 全局作用域

跟局部作用域相对应的是全局作用域。如果变量或者函数不在任何局部作用域内，就说这个变量或者函数在全局作用域中，称为全局变量或者全局函数。全局作用域从变量或者函数定义时开始到整个源文件结束，换句话说，全局作用域就是整个源文件范围。例如，在一个源文件中，可以这样使用全局变量和全局函数：

```
// 定义一个全局变量
int gN;

// 定义一个全局函数
void GlobalFunc(int a, int b)
{
    for( gN = 0; gN < 10; ++gN )    // 访问全局变量
    {
        //...
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    gN = 3;                        // 访问全局变量
    GlobalFunc( 2, 4 );            // 调用全局函数
    //...
    return 0;
}
```

在这个例子中，因为在源文件的开始定义的变量 gN 没有在任何局部作用域，所以它的作用域就是全局作用域，其作用范围是整个源代码文件，因此可以在后面的任何位置访问这个变量。而 GlobalFunc() 是一个全局函数，在其声明之后可以在同一个源文件的任何位置调用这个函数。

在 7.3.1 小节中曾经介绍过，C++ 程序往往被分解为多个源文件和头文件，但是全局作用域内的变量或者函数只在单个源文件范围内可见。如果想在多个源文件中使用某个源文件定义的全局变量应该怎么办？很简单，“extern”关键字可以找到定义在其他源文件中的全局变量。我们知道，如果想使用某个变量或者函数，则必须事先进行声明。当想使用某个已经在其他源文件中声明的全局变量或全局函数时，可以在这个源文件中对变量或者函数加上“extern”关键字进行修饰，对变量或者函数重新进行声明，表示这是一个扩展的声明，编译器会在其他源文件中查找这个变量或者函数的具体定义，从而使用同一个变量或者函数。例如，在 Global.cpp 文件中定义一个全局变量和全局函数。


```
// Global.cpp: 定义全局变量和全局函数
#include "StdAfx.h"
// 全局变量
int gTotal = 0;
// 全局函数
int Add( int a, int b )
{
    return a + b;
}
```

如果想在另外一个源文件中使用这个全局变量和全局函数，就需要用“extern”关键字对它们重新进行声明，然后才可以开始使用。

```
// HelloWorld.cpp: 使用全局变量和全局函数
// 在变量声明前加上“extern”关键字，重新声明全局变量
extern int gTotal;
// 在函数声明前加上“extern”关键字，重新声明全局函数
extern int Add( int a, int b );

int _tmain(int argc, _TCHAR* argv[])
{
    // 使用全局变量和全局函数
    gTotal = Add( 2, 3 );

    return 0;
}
```

这样，通过“extern”关键字，变量或者函数就可以冲出单个源文件，走向整个项目。可以在多个源文件之间共享全局变量和全局函数，实现真正的“全局”作用域，如图 7-9 所示。

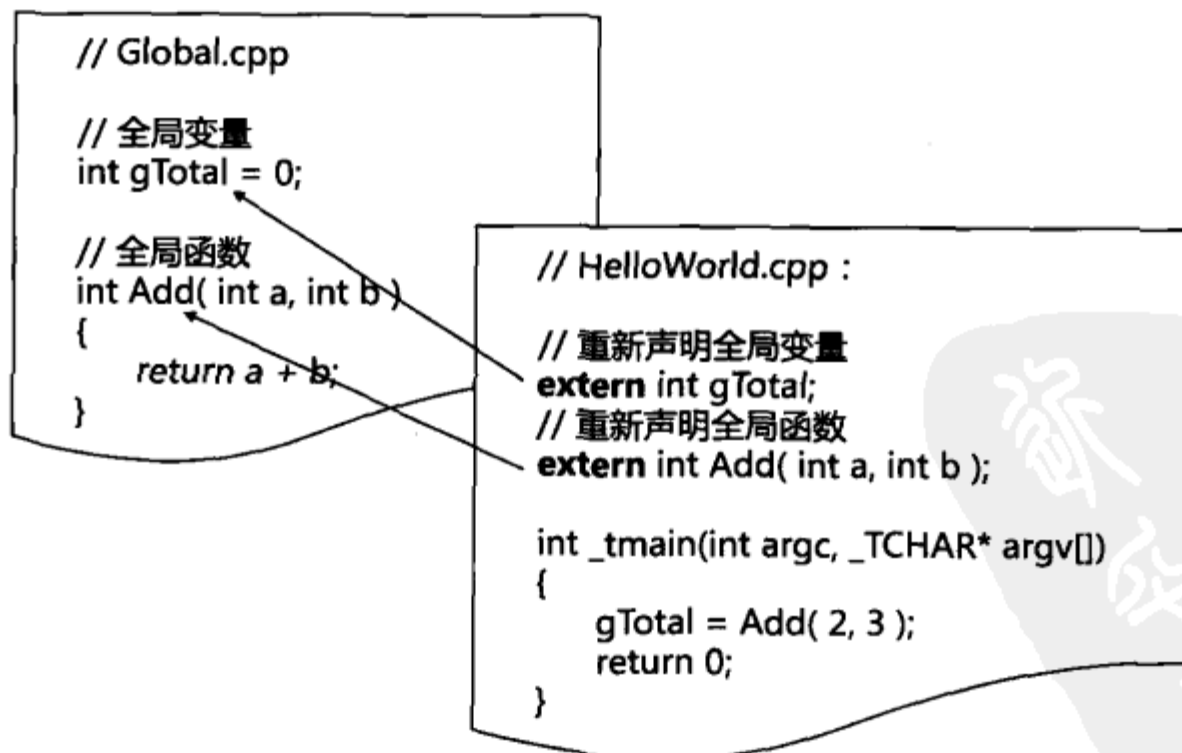


图 7-9 多个源文件共享全局变量和全局函数

7.3.4 编译预处理

到现在为止，已经基本了解了如何去搭建一个比较复杂的 C++ 程序框架，在整个框架之下可以开始代码的编写了。然而，编写的代码是否就是最终参与编译器编译的代码呢？不是的，预编译程序还在编写的代码和最终的源代码之间做了手脚。编译器在对源程序进行编译之前，首先要由编译预处理程序对程序文件进行预处理，这个过程称为编译预处理。可以通过在源代码中加入一些特殊的预编译指令来改变编译的过程，甚至改变程序的源代码结构和内容，达到灵活处理源文件的目的。下面介绍几种最常用的预编译指令。

1. #include 指令

在前面的学习中，已经久闻 #include 指令的大名了，它可以用来将一个文件嵌入当前位置，实现多个源文件共享同一个文件，共用其中的内容。更多时候，用 #include 指令来嵌入一个头文件，实现对声明在其中的变量或者函数的引用。#include 指令的语法格式如下：

```
#include <文件名>
#include "文件名"
```

注意到，这里有两种引入文件名的方式。其中，“<>”表示按照标准方式在编译器指定的目录下搜索这个文件，而“””则表示先在当前目录下搜索，如果当前目录下没有这个文件，然后按照标准方式搜索。一般来说，用“<>”来引入系统提供的头文件，比如 iostream 等，而使用“””来引入自己创建的、放置在当前项目文件夹下的头文件。例如：

```
// 引入标准头文件
#include "stdafx.h"
// 引入标准程序库中的头文件
#include <iostream>
// 引入自己创建的头文件
#include "Global.h"
```

2. #define 指令和#undef 指令

#define 指令可以用来定义一个符号常量或者宏，这个符号常量往往要在以下介绍的条件编译指令中使用到。

#undef 的作用是删除一个由 #define 定义的符号常量或者宏。例如：

```
// 定义一个符号常量_DEBUG
#define _DEBUG
```

3. #if 等条件编译指令

一些复杂的 C++ 程序往往需要根据设置或者外部环境条件的不同而编译成不同的版本，比如，最常见的可以分成 Debug 版本和 Release 版本、带日志输出的版本和不带日志输出的版本、ANSI 版本和 UNICODE 版本等，这时就可以利用条件编译指令，根据不同的条件改变

参与编译的源代码，从而将同一份程序源代码编译成不同的程序版本。条件编译指令常见的形式如下：

```
// 条件编译指令开始
#if 常量表达式
    // 当常量表达式为 true 时，本段程序代码参与编译，
    // 否则不参与编译
    程序代码
#endif          // 表示条件编译结束

#ifdef 标识符
    程序段 1      // 如果定义了标识符，则编译程序段 1
#else
    程序段 2      // 如果没有定义标识符，则编译程序段 2
#endif

#ifndef 标识符
    程序段        // 如果没有定义标识符，则编译此程序段
#endif
```

条件编译指令是在开发实践中经常用到的预编译指令，例如下面这个例子：

```
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
```

这段代码使用了“#ifdef”预编译指令，它表示如果定义了_DEBUG 标识符，相当于在编译 Debug 版本时用“#define”指令定义这个标识符，就会编译中间的程序代码，为程序增加两个函数用于辅助调试。而在编译 Release 版本的时候，并没有定义_DEBUG 标识符，就不编译这两个函数，从而减小程序的体积。

4. #pragma once 指令

该指令的作用是指定当前文件在编译时只包含一次，这样就可以避免同一个文件被多次引入而导致数据类型或者函数等的重复定义，同时减少了编译的时间。这个指令通常使用在头文件中，以防止头文件重复多次引入。因为头文件中加入“#pragma once”指令之后，预处理程序在处理源文件时，第一次遇到“#include”指令引入某个头文件，这个头文件将打开并包含进来；如果再次遇到相同的“#include”指令，这个头文件就不会重复包含进来，以此达到防止一个头文件重复多次引入的目的。例如一个标准的 stdafx.h 头文件：

```
// stdafx.h : 标准系统包含文件的包含文件，
// 或是经常使用但不常更改的
// 特定于项目的包含文件
//...

// 使用“#pragma once”指令，防止头文件多次引入
```

```
#pragma once
// ...
```

7.4 高手是这样炼成的

见过了 C++ 世界的奇人异事之后，还是觉得不过瘾。好，下面介绍的都是高手们的终极必杀技，修炼时务必小心谨慎！

7.4.1 用宏定义化繁为简

宏定义又称为宏替换，简称“宏”。在 C++ 世界中，使用“#define”指令来定义一个宏：

```
#define 标识符 字符串
```

其中，标识符就是所谓的符号常量，也称为“宏名”。当定义好宏之后，就可以在程序代码中使用这个宏来代替宏定义中的字符串。例如：

```
// 定义一个宏
#define PI 3.14159

// 利用定义的宏计算圆的面积
double fR = 5.0f;
double fArea = PI * fR * fR;
```

当预编译程序对程序代码进行预处理时，如果遇到代码中使用了宏，就会将宏展开，也就是将宏名替换为宏定义中的字符串。所以，宏展开后，这段代码实际上成为：

```
// 宏展开后的代码
double fArea = 3.14159 * fR * fR;
```

这里可以看到，宏的本质就是“替换”，也就是“偷梁换柱”：偷去程序代码中的宏名，换上宏定义中的字符串。虽然在程序代码中使用的是宏，但是经过预处理后最终参与编译的代码却是“替换”后的代码。所以，要理解别人使用宏的代码，先要进行“替换”，同样，如果代码中有需要进行“替换”的地方，比如可以将这些长长的字符串常量替换为一个简短的宏，或者利用宏给某个字符串一个更有意义的名字，都可以使用宏，如图 7-10 所示。

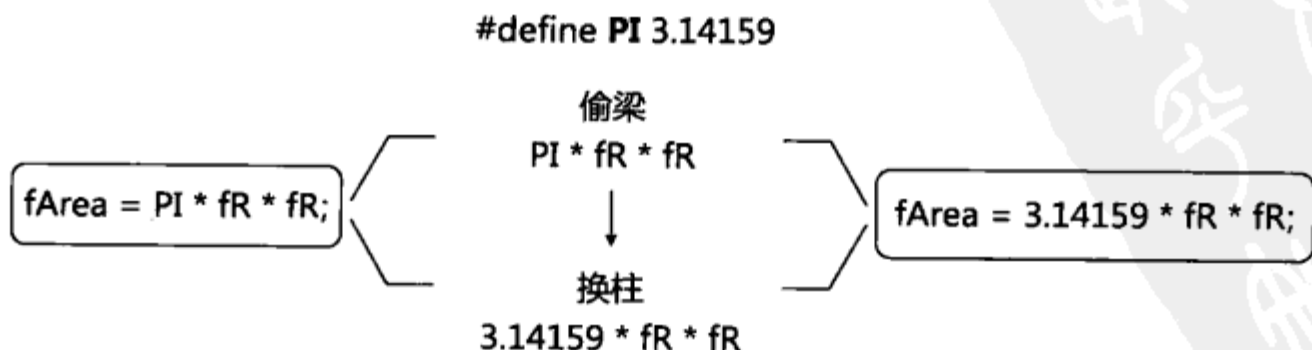


图 7-10 宏的“偷梁换柱”

除了这种不带参数的宏之外，还可以在宏定义中加入参数，让宏的使用更加灵活。带参数的宏定义的语法格式如下：

```
#define 标识符(参数列表) 字符串
```

其中，参数列表列出的就是这个宏的所有参数。跟函数参数不同，宏的参数并没有类型的概念，更多的只是一个替代，用作宏展开时使用相应的实际参数替换参数列表中的参数。对带有参数的宏进行宏展开时，不仅要对其标识符作字符串替换，还必须作参数的替换。例如：

```
// 定义一个带参数的宏，取得两个数中比较大的一个数
#define MAX( a, b ) ((a) > (b) ? (a) : (b))
```

```
// 使用宏取得 2 和 4 中较大的一个数
int nMax = MAX( 2, 4 );
```

当进行宏展开时，这个宏首先进行字符串的替换，这行代码变为：

```
int nMax = a > b ? a : b;
```

然后，使用宏的实际参数替换宏定义中的参数，这行代码最终成为：

```
int nMax = 2 > 4 ? 2 : 4;
```

经过一系列的替换，终于看到了这行代码的本来面目，原来就这么简单啊！

使用宏可提高程序的通用性和易读性，减少不一致性、输入错误和修改量，所以，高手们都喜欢用它。

7.4.2 用 typedef 定义类型的别名

在武林中，为了便于记忆和传颂，有给大侠们取外号的习惯，所以这才有了“及时雨——宋江”和“豹子头——林冲”。在 C++ 世界中，也有给一些数据类型取别名的习惯，C++ 还专门提供了一个给人取别名的关键字——typedef，自号“千面娇娃”。

在 C++ 世界中，常常使用 typedef 关键字给某些比较复杂的难以书写的类型取别名，然后在程序中使用类型的别名来指代这个类型，这样可以达到简化程序代码的目的，从而增加程序的可读性及标识符的灵活性。就像一个武林人士取外号是为了唬人那样，C++ 世界中用 typedef 定义的类型别名是为了偷懒。在 C++ 世界中，用 typedef 定义类型别名的语法格式如下：

```
typedef 数据类型 数据类型的别名
```

typedef 的使用非常简单，在 typedef 关键字之后，分别跟上数据类型和相应的别名就定义了这个数据类型的别名，在接下来的程序中就可以使用这个别名来指代这个数据类型了。例如，觉得指针类型的书写比较烦琐时，可以使用 typedef 为指针类型定义一个简单易记的别名。

```
// 为指针类型 int* 定义一个别名 PINT
typedef int* PINT;
```

有了这个简单的别名，就可以用这个简单的别名来指代指针类型。

```
// 定义一个 PINT 类型的变量，实际上就是 int* 类型的变量
PINT pInt = NULL;
```

使用别名之后的程序代码，是不是书写起来更加简单，同时也更加简洁易懂呢？是的。所以，在必要的时候使用 typedef 为类型取一个合适的别名，让类型在不同的情况下以不同的面目出现也是很重要的。为什么高手们写的代码读起来如沐春风？“千面娇娃” typedef 功不可没。

在 C++ 世界中，typedef 关键字还可以应用在多个方面，充分发挥出“千面”的威力。

1. 定义一种类型的别名

typedef 是为类型定义的别名，而不只是简单的宏替换。这一点在同时声明指针类型的多个变量时特别有用。例如，想声明两个 int 类型的指针，直观地，可能会这样写：

```
// 尝试声明两个 int 类型的指针
int* pInt1, pInt2;
```

可是，遗憾的是，得到的却是一个 int 类型的指针 pInt1，而实际上 pInt2 却是一个 int 类型的变量！难道就不能在同一行代码中同时定义多个指针吗？对于万能的 C++ 语言来说，没有什么是不可能的。使用 typedef 定义一个 int 指针类型的别名，就可以轻松做到。

```
// 为指针类型 int* 定义一个别名 PINT
typedef int* PINT;
// 同时定义多个指针类型变量
PINT pInt1, pInt2;
```

这时，PINT 已经成为一种新的类型，它可以同时定义多个 PINT 新类型的变量，而这种新类型本质上还是 int 指针类型。

2. 定义与平台无关的类型

某些数据类型的实现是跟具体的平台相关的，可能在某个平台上有某种数据类型，但是在另外的平台上这个数据类型并不存在。这就给 C++ 代码的移植造成了困扰，我们不可能为了不同的平台而更改代码，使用不同的数据类型。这时，可用 typedef 来定义一个与平台无关的新数据类型，用这个新数据类型来代替跟平台相关的数据类型就可以很好地解决这个移植问题。例如，可以定义一个叫 REAL 的浮点数类型。在支持 long double 的目标平台上，可以将其作为 long double 的别名，表示最高精度的浮点数。

```
typedef long double REAL;
```

当要将代码移植到一个不支持 long double 的平台时，可以把这个定义修改如下：

```
typedef double REAL;
```


这样，就可以在代码中统一使用 REAL 类型，表示目标平台所支持的最高精度的浮点数类型，而不管这个平台是否支持 long double 或者 double。

另外，因为 typedef 是定义一种类型的新别名，而不是简单的字符串替换，所以它比宏来得稳健，虽然使用宏有时也可以完成以上的任务。

3. 为复杂的声明定义简单的别名

请看下面这行代码：

```
int* (*pFunc)(int, char*);
```

你能一下看出这行代码到底在搞什么吗？

如果能，那么恭喜你跳过这个章节，开始后面的旅程了；如果不能，那么还是一起好好学习如何利用 typedef 来简化这堆代码吧！

实际上，这行代码所定义的是一个函数指针 pFunc，这个函数指针所能够指向的函数的返回值类型是 int*，两个参数分别是 int 类型和 char* 类型。如果只是定义一个既难懂又难写的函数指针，还可以勉强接受；如果要定义多个这种类型的函数指针，那么多次书写这么复杂的声明语句，恐怕要“撞墙”了。好在 typedef 可以为这种复杂的类型定义一个简单的别名，所以，“撞墙”还是留给那些不会使用 typedef 的人吧！

```
// 定义函数指针类型为 PFUNC
typedef int* (*PFUNC)(int, char*);

// 使用 PFUNC 定义多个函数指针变量
PFUNC pFunc1, pFunc2;
```

从此，这个世界清静了！

7.4.3 用 const 保护数据

在程序中什么最重要？数据最重要！可以说，整个程序都是围绕数据在打转，从数据的输入到数据的输出、从数据的处理到数据的转移，程序就是为了数据而生的。程序中的数据如此重要，谁也不想让自己的数据未经授权就被随意修改而导致最后的结果出错。所以，才有了在 6.2.6 小节中介绍过的通过定义类成员的访问级别来防止外界非法地访问类中的数据，以此保护数据的安全性。除此之外，C++ 语言还提供了专门的“固若金汤”大法——const 关键字来保护数据，以防止数据被非法修改。大家知道，我们是使用各种类型的变量来保存数据的，如果想保护某个变量的值，使之不被修改，可在声明这个变量的时候，在数据类型前加上 const 关键字进行修饰就可。其语法格式如下：

```
const 数据类型 变量名;
```


例如，可以这样来保护数据：

```
const double PI = 3.14159;
PI = 3.14;    // 想偷工减料？有 const 保护，这是行不通的
```

经过 const 修饰，变量就具有了 const 属性。当在程序中尝试修改这个变量的值时，编译器老大就会站出来说话：

“嘿，这个变量是我罩着的，你小子别想动它！”

使用 const 关键字并靠上编译器这位老大，还有什么好担心的呢？

在实际应用中，const 关键字的用法很多，大致有以下几种情形。

1. 使用 const 代替#define 定义常量

比如要定义 PI 常量，可以采用下面的两种方式：

```
// 定义宏 PI
#define PI 3.1415926
// 定义常量 PI
const double PI = 3.14159
```

这两种形式在语法上都是合法的，但是第二种形式要比第一种形式好。这是因为如果使用#define 定义宏 PI，PI 会在编译之前被预处理程序替换成具体的数字，宏的名称不会出现在符号表中，所以会给调试带来一定麻烦，可能会遇到一个不知道从何而来的数字。而使用 const 定义常量，既可以保证 PI 值的一致性，又便于调试，同时还可以进行类型检查，借助编译器来减少错误的发生。

2. 使用 const 表示常量

使用 const 关键字，编译器可以检测出对常量的非法操作，以防止非法修改常量。const 关键字常见的常量声明方式如下：

```
// 声明一个整型常量并赋初值为 1
const int number = 1;
// 声明一个 Teacher 类型的常量对象
const Teacher MrChen;
// 声明一个常量整型指针，指针所指向的变量的值不能修改，也就是一个常量
const int* pNumber;
// 声明一个常量整型指针，意义同上
int const * pNumber;
// 声明一个整型常量指针，指针不能修改
int * const pNumber = &number;
// 声明一个常量整型常量指针
// 指针和指针所指向的变量值都不能修改
const int * const pNumber = &number;
// 声明一个常量整型引用
```

```
const int & number = number;
```

这里要特别注意的是，第二种常量整型指针（`int const * pNumber`）和第三种整型常量指针（`int * const pNumber = &number`）的区别。因为 `const` 关键字位置的不同，导致这两种形式表达的意义差别很大。为了弄清楚这两种形式的差别，可以以 “*” 为界，把声明语句分割成两个部分，如图 7-11 所示。

```
int const  | pNumber;  
int        | const pNumber = &number;
```

图 7-11 使用 “*” 分割常量整型指针和整型常量指针

经过这样的分割后，就可以很容易地总结出规律：如果 `const` 在 “*” 的左边，则表示 `const` 修饰的是 `int`，这个指针指向的 `int` 变量的值不能修改，而指针本身的值是可变的；如果 `const` 在 “*” 的右边，则表示 `const` 修饰的是指针，这个指针的值不能在声明后修改，所以在声明这样的指针时必须赋初值，而这个指针所指向的 `int` 变量的值是可变的。

3. 表示函数的输入、输出参数

因为 `const` 所修饰的变量的不可修改性，所以常常给函数的传入参数加上 `const` 修饰，表示这只是一个传入参数，在整个函数内部不能被修改。例如：

```
bool AutoResizeRect( CRect& destRect,      // 传出参数，可以修改  
                    const CRect& srcRect ) // 传入参数，不可以修改  
{  
    // ...  
}
```

其中，使用 `const` 关键字把表示原始数据的传入参数保护起来，防止这个数据在函数中被修改而影响函数外的数据。

4. 修饰类成员函数

在声明类的成员函数时，如果在末尾加上 `const` 修饰，则表示在这个成员函数内不得改变该对象的任何数据。这种模式常用来表示“对象数据只读”的访问模式，也就是说，这是一个只读的成员函数，也称为查看函数。如果成员函数末尾没有 `const` 修饰，则表示在这个成员函数内可能会对对象的数据进行修改，所以这种没有 `const` 修饰的成员函数也称为变更函数。例如：

```
class ConstDemo  
{  
public:  
    // 获取类中的数据  
    // 在函数后添加 const 修饰，表示这是一个查看函数  
    char GetValueAt(int nIndex) const  
    {  
        // 对类中的数据进行意外的修改  
    }
```

```

        // 会导致一个编译错误
        m_data[3] = 'A';
        return m_data[nIndex];
    }

    // 设置类中的数据
    // 在函数后未添加 const 修饰，表示这是一个变更函数，
    // 它将有可能修改类中的数据
    void SetValueAt(int nIndex, char cNewValue)
    {
        m_data[nIndex] = cNewValue;
    }
protected:
    char m_data[10];
};

```

在这段代码中，GetValueAt()函数使用 const 修饰，表示这是对类的一个只读访问。如果试图在 GetValueAt()函数内部修改这个类的数据，编译器就会产生一个编译错误，这可以帮助我们检查程序代码中的对数据的非法修改。

const 关键字除了可以发现成员函数中对数据的非法修改之外，当在一个 const 修饰的常量对象上调用变更成员函数、尝试对常量对象进行修改时，编译器也会发现这样的非法修改错误。例如：

```

// 定义一个只读的 ConstDemo 类型的常量对象 constObject
const ConstDemo constObject;
// 调用常量对象的查看成员函数进行只读访问是合法的
char cValue = constObject.GetValueAt(0);
// 调用常量对象的变更成员函数，尝试修改常量对象的数据，
// 这将导致一个编译错误
constObject.SetValueAt(0, 'J');

```

当编译器编译这段代码时，会发现变更成员函数 SetValueAt()对常量对象 constObject 的非法修改，从而产生一个编译错误，提示我们修改代码。恰当地使用 const 修饰类的成员函数，可以防止对类数据的意外修改，从而起到保护数据的作用。

施展开 const 这个“固若金汤”大法，现在谁也别想动数据了！

第3篇

攀登〇++世界的高峰



用 STL 优雅你的程序

在见识了 C++ 世界的奇人异事之后，我们自然是功力倍增、信心满满，遇到什么事都想用 C++ 语言来解决，以期在众人面前显摆自己的功力是如何如何高深。正好，那天老板对前面的工资统计程序提出了新需求：统计员工中拿高工资的人数，也就是工资高于 1 000 元的人数。看到这个新需求，我们一定会在心中暗暗发笑：这还不简单，小菜一碟啊！很快，就有了自己的实现方法。

```
// 统计高工资人数
const int MAX_COUNT = 10000;
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义保存员工信息的数组
    int arrSalary[MAX_COUNT];
    // 当前员工个数
    int nCount = 0;
    // 用户输入...

    // 高工资的员数
    int nTotal = 0;
    // 循环遍历数组进行统计
    for( int i = 0; i < nCount; ++i )
    {
        if( arrSalary[i] > 1000 )
            ++nTotal;
    }
    // 结果输出...
}
```

正当我们暗自得意、想要把程序拿到老板面前去邀功请赏时，这时过来一位白衣飘飘的“绅士”，对代码看了两眼，嘴里说出一句话来：“粗鲁！四肢发达的程序员才这么干！”

自己非常满意的代码居然被人说成是“粗鲁”的代码，心中难免有些不爽，转头想找“他”理论理论，没想到“他”已经飘然走远了。为了以后好找“他”算账，于是赶紧问：“请问大侠大名？”“标准模板库，你也可以叫我 STL。”

听到这里，我们不由得暗暗叹道：原来“他”就是传说中的优雅绅士 STL 啊，今日得见，果然名不虚传、气度非凡啊。心中害怕错过这个结识的机会，于是连忙说道：

“STL 大侠，别走啊，我们交个朋友吧！”

8.1 跟 STL 做朋友

要想跟 STL 做朋友，就得先从 STL 开始了解。

8.1.1 算法 + 容器 + 迭代器 = STL

STL，即标准模板库，是一个具有工业强度的、高效的 C++ 程序库。据 C++ 世界的“老人们”说，STL 最初诞生于惠普实验室，它是由 Alexander Stepanov、Meng Lee 和 David R. Musser 在惠普实验室工作时所开发出来的，后来经过不断地发展形成不同的版本。现在，它已经容纳于 C++ 标准程序库（C++ Standard Library）中，是 ANSI/ISO C++ 标准中最新的也是极具创新性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法，为广大 C++ 程序员提供了一个可扩展的应用框架，高度体现了软件的可复用性。我们不仅可以直接从中获得极大的开发便利，也可以通过继承现有类，自己编制符合接口规范的容器、算法、迭代器等方式对之进行扩展。

从广义上讲，STL 主要分成三大核心部分：算法（algorithm）、容器（container）和迭代器（iterator）。除此之外还有容器适配器（container adaptor）、函数对象（functor）等。几乎 STL 的所有代码都采用了模板类和模板函数的方式，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。下面先对 STL 的三大核心部分做一个简单了解。

1. 容器

在程序中，我们使用各种数据结构来对数据进行管理。虽然常用的数据结构数量有限，但是可以重复一些为了实现向量、链表等基础数据结构而编写的代码，这些代码都十分相似，它们只是为了适应不同的数据变化而在细节上有所出入。STL 容器就提供了这样的方便，它允许重复利用已有的实现，从而构造自己的特定类型下的数据结构。通过设置一些模板类，STL 容器提供支持最常用的数据结构，这些模板的参数允许指定容器中元素的数据类型，将许多重复而乏味的工作简化。

2. 算法

算法是应用在容器上的以各种方法处理容器中内容的行为或功能。STL 包含了很多计算机科学上的通用算法，比如，算法 `for_each` 将为指定容器中的每个元素调用指定的函数，`stable_sort` 以所指定的规则对容器中的元素进行稳定性排序等。这些都是我们在日常开发活动中经常会用到的算法。算法本身与操作的数据结构和类型无关，因此可以在从简单数组到高度复杂容器的任何数据结构上使用。这样一来，只要我们熟悉了 STL，就可以大大简化许多

代码，只要通过调用一两个算法模板，就可以完成所需要的功能，从而大大提升开发效率。

3. 迭代器

如果容器用于容纳数据、算法用于处理数据，那么迭代器就像胶水一样将算法和容器紧密地结合在一起——算法通过迭代器来定位和操控容器中的元素。如果没有迭代器，算法和容器就无法相互作用。事实上，每个容器都有自己的迭代器，只有容器自己才知道如何访问自己的元素，这使得迭代器就像一个指向容器中元素的普通指针一样。

正是算法、容器和迭代器这三个核心部分相互配合、相互作用，使得 STL 成为了一个有机的整体，如图 8-1 所示。



图 8-1 STL 的三大核心部分

8.1.2 在程序中使用 STL

因为 STL 已经是 C++ 标准库的一部分，所以无须其他额外操作就可以在 C++ 代码中直接使用 STL。根据功能的不同，STL 被组织在多个头文件和名字空间中，要想在程序中简单使用 STL，只需要引入相应的头文件，使用对应的名字空间即可。其中，常用的 STL 头文件和名字空间如表 8-1 所示。

表 8-1 STL 的常用头文件和名字空间

头 文 件	名字空间	描 述
<deque>	std	双端队列容器，它是一个由连续存储的指向不同元素的指针所组成的数组
<queue>	std	队列容器，按照先进先出的规则排列容器中的数据
<stack>	std	堆栈容器，按照后进先出的规则排列容器中的数据
<vector>	std	动态数组容器，连续存储的容器中的元素，它是最常用的非关联容器
<map> <multimap> <unordered_map> <unordered_multimap>	std	映射容器，由{键，值}对组成的集合，以某种作用于键值对上的谓词排列，它是最常用的关联映射容器。其中，map 容器中的键值对是一一对应的关系，而 multimap 容器中一个键可以对应多个值。 作为映射容器，map 和 multimap 由来已久，其底层由红黑树实现，而 unordered_map 和 unordered_multimap 是最新的 C++0x 标准新添加入 STL 的映射容器，其底层由哈希表实现

头文件	名字空间	描述
<code><set></code> <code><multiset></code> <code><unordered_set></code> <code><unordered_multiset></code>	std	集合容器，由节点组成的红黑树，每个节点都包含一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序。其中， <code>unordered_set</code> 和 <code>unordered_multiset</code> 是 C++0x 标准新加入 STL 中的集合容器，其底层由哈希表实现
<code><algorithm></code>	std	它是所有 STL 头文件中最大的一个（尽管它很好理解），它是由很多模板函数组成的，它们相互独立构成 STL 中的通用算法，包括比较、交换、查找、排序等
<code><functional></code>	std	定义了一些模板类，用以声明函数对象
<code><string></code>	std	字符串类
<code><regex></code>	std	正则表达式，用于对字符串进行处理
<code><memory></code>	std	其中定义了跟内存操作相关的组件，例如智能指针等

既然 STL 的使用这么简单，就不妨将 STL 应用到程序中，让“粗鲁”的工资统计程序也变得优雅起来。

```
#include "stdafx.h"
#include <iostream>
// 引入 STL 相关组件的头文件
#include <vector>      // 向量容器
#include <algorithm>   // 通用算法
using namespace std;  // STL 组件所在的名字空间

int _tmain(int argc, _TCHAR* argv[])
{
    // 使用 vector 容器保存用户输入的数据
    vector<int> vecSalary;

    cout<<"请输入工资数据，0 表示输入结束"<<endl;

    // 用户输入
    int nSalary = 0;
    do
    {
        cin>>nSalary;
        if ( 0 == nSalary )
            break;

        // 将用户输入的数据保存到容器中
        vecSalary.push_back(nSalary);
    } while(true);

    // 使用通用算法 count_if 统计 vecSalary 容器中大于 1000 的元素个数
    int nTotal = count_if( vecSalary.begin(), vecSalary.end(),
        bind2nd( greater<int>(), 1000 ) );
```

```
// 结果输出
cout<<"工资超过 1000 的员工数是: "<<nTotal<<endl;

return 0;
}
```

这段程序使用了 STL 中的 vector 容器来管理输入的工资数据,也使用了 STL 中的 count_if() 算法来统计容器中满足条件的元素个数,通过容器和算法的配合使用,轻松简单地完成了工资统计程序。STL 的使用就这么简单。

8.1.3 STL 到底好在哪里

大家可能会问,“虽然 STL 的使用简单,但是没有看出使用 STL 到底好在哪里啊?”不怕不识货,就怕货比货。通过简单的比较,可以明显看出本章开始部分所写的“粗鲁”的程序和使用 STL 修改后的“优雅”的程序两者之间的差别:从内存空间的使用上看,“粗鲁”的程序使用数组来保存用户输入,并且很快就将数组元素的个数定义为 10 000,难怪被人嘲笑为“粗鲁”了。我们知道,数组的容量需要事先定义,且容量在其生命周期内是固定不变的。如果容量定义太多,就会造成空间的浪费;如果容量定义太少,又会不够用,不利于程序的扩展。而在“优雅”的程序中,使用 STL 中的 vector 容器来保存用户输入。vector 容器可以根据用户的输入动态地增长,这样既可以保证内存空间的合理利用,又可以让程序具有很大的可扩展性,处理很大范围内的数据。

从开发的效率和可维护性上看,在“粗鲁”的程序中,使用了一个 for 循环来统计数组中工资大于 1 000 元的员工人数,这种方法虽然管用,但是显得很生硬。首先,需要知道数组中已经保存的工资条数,这就需要增加一个额外的变量来记录工资条数。另外,如果想对其进行扩展,统计其他工资范围内的员工个数,比如工资小于 2 000 元的员工条数,那么还需要写另外一个 for 循环,这样使得整个程序显得纷繁复杂,降低了开发效率,同时也增加了后期的维护成本。如果使用 STL 中的通用算法,再配合使用 STL 中的容器,就可以很好地解决这些问题。使用 count_if 算法,可以对容器中的元素进行各种统计,而其中的统计函数则是用于专门负责统计的规则。既可以统计工资大于 1 000 元的员工人数,也可以统计其他工资范围的员工人数,这样不仅可以提高开发效率,而且增加了程序的可扩展性。

总的来说,STL 有以下这些优点:

- 封装很多常用的数据结构,并为这些数据提供一致的操作方式;
- 提供很多常用的通用算法,并能够很方便地排序、搜索容器中的数据等,提高开发效率;

- 调试程序时更加安全和方便；
- 支持跨平台，使用 STL 编写的 C++ 代码可以轻松地移植到其他平台中。

STL 有如此多的优势，使得 STL 俨然成为 C++ 世界中的一位“优雅绅士”，成为每个 C++ 语言初学者都想结识的对象。

8.2 用模板实现通用算法

在程序中大家都愿意使用 STL，跟 STL 做朋友，那是因为 STL 很方便实用，大大提高了开发效率。大家可否知道，当初 STL 是由一个不太勤快的程序员发明的。

一天，这个不太勤快的程序员接到一项任务，要求写一个比较两个整数大小的函数。这个任务很简单啊，程序员很快就有了如下的实现：

```
// 比较两个整数的大小
int max(int a,int b)
{
    return a > b ? a : b;
}
```

但是第二天，老板又给了他一项新的任务：要求写一个比较两个浮点数大小的函数。不勤快的程序员心想，幸亏我学过函数重载，要不然还麻烦了。于是想都不想，直接 copy 了第一天的代码，修改后就有了比较两个浮点数大小的函数：

```
// 比较两个浮点数的大小
float max(float a, float b)
{
    return a > b ? a : b;
}
```

可第三天新任务又来了：要求写一个比较字符串大小的函数。这下程序员的头大了，整数、浮点数、字符串、自定义类型……这无穷无尽的 max 函数何时是个头啊。要是可以像写工作总结一样，有个模板就好了。这样可以把 max 函数做成一个模板，针对不同的数据类型直接替换就可生成不同版本的 max 函数，多省事啊。一个不太勤快的程序员的偷懒想法使得 C++ 世界中模板的概念诞生了，STL 开始萌芽。

就像利用文档模板实现文档格式和内容的重用一样，模板是实现代码重用的一种重要机制，它可以实现类型的参数化，即把类型定义为参数。算法是一个模板，通过将具体的数据类型带入模板中以实现具体的针对特定数据类型的算法，从而实现真正的代码可重用性，STL 也正是借助模板的威力而构建起来的。

在 20 世纪 70 年代末，Alexander Stepanov 第一个发现算法不依赖于数据结构的特定实现，而仅和数据结构的一些基本语义属性相关。这些属性表达了一种能力，比如可以从数据结构的一个成员取得下一个成员、从头到尾遍历结构中的元素等，比如排序算法不关心元素是存放在数组中或是线性表中。Alexander Stepanov 研究之后发现，一些通用算法可以用一种抽象的方式实现，而且不会影响效率。正是他的这个发现，成为 STL 的思想源起。

1985 年，Alexander Stepanov 开发了基本 Ada 库。由于当时 C++ 开始流行，于是人们要求他在 C++ 中也这样做，但直到 1987 年，模板(template)在 C++ 中还未实现，所以他的工作推迟了。1988 年，Alexander Stepanov 到惠普实验室工作，并在 1992 年任命为一个算法项目的经理。在此项目中，Alexander Stepanov 和 Meng Lee 编写了一个巨大的库——标准模板库，意图定义一些通用算法而不影响效率。这个标准模板库成为 STL 的雏形。

1994 年 7 月 14 日，ANSI/ISO C++ 标准化委员会采纳 STL 为草案标准。现在，各个 C++ 编译器都支持 STL，STL 已经并将继续影响 C++ 的开发方法。有了 STL，程序员可以写更少且更快的代码，从而把精力集中在问题解决上，而不必关心底层的算法和数据结构了。

8.2.1 函数模板

在第 5 章中，我们将一个函数比作一个箱子。通常，箱子都是专用的，比如装衣服的箱子只能用来装衣服，装零食的箱子只能用来装零食。同理，比较 int 类型数据的 max() 函数只能用来比较 int 类型数据，而比较 float 类型数据的 max() 函数也只能用来比较 float 类型数据。但是，借助模板可以创建一种万能箱子——函数模板，它可以用来装各种东西、处理各种类型的数据。比如当编译器发现一个函数模板的调用后，将根据实参的实际数据类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数，称该重载函数为模板函数。根据参数类型的不同，一个函数模板可以随时变成各种不同的重载函数，从而实现对各种数据类型的处理。

在 C++ 世界中，有很多算法需要能够同时处理各种不同的数据类型，比如一个比较大小的算法，既要能够比较整型数，也要能够比较字符串。虽然这些算法处理的数据类型不同，但是算法本身是相同的，这种情况下就可以使用函数模板来简化工作，实现代码的复用。一般来讲，函数模板可以用来创建一个通用功能的函数，以支持各种不同的数据类型，简化重载函数的设计。函数模板的声明非常简单，其语法格式如下：

```
template <typename 标识符>
返回值类型 函数名(形参表)
{
```

```

    // 函数体
}

```

其中，用 `typename` 定义的标识符就是函数模板中抽象的数据类型。在函数模板中，可以使用这种抽象的数据类型作为模板来代替实际的数据类型。以 `max()` 函数为例，看看如何使用函数模板来简化函数重载的工作。

```

// 自定义的比较函数模板
// T 就是函数模板的参数
template <typename T>
T mymax( const T a, const T b )
{
    return a > b ? a : b ;
}

```

可以看到，在 `max()` 这个函数模板中，使用了 `T` 这种抽象的数据类型来代替实际的数据类型。这种抽象数据类型就像函数模板的一个参数，当使用函数模板时，这个参数会被定义为实际调用时的某种具体数据类型，然后利用函数模板生成具体的重载函数。例如：

```

int _tmain(int argc, _TCHAR* argv[])
{
    // 使用函数模板，比较整数
    int nA = 2;
    int nB = 5;
    wcout<<nA<<"和"<<nB<<"中比较大的是"
        <<mymax(nA,nB)<<endl; // 动态生成模板函数 int mymax(int,int)

    // 使用函数模板，比较浮点数
    float fA = 2.2;
    float fB = 5.5;
    wcout<<fA<<"和"<<fB<<"中比较大的是"
        <<mymax(fA,fB)<<endl; // 动态生成模板函数 float mymax(float,float)

    return 0;
}

```

在这段程序中，我们分别使用了不同的数据类型调用 `mymax()` 函数模板，函数模板根据不同的数据类型生成不同的模板函数，最终利用一个函数模板完成对多种数据类型的处理。很明显，使用函数模板后，减少了对函数的重载，程序的代码量大大减少，也方便了复用，这就是函数模板的优势。

大家一定会觉得函数模板可以根据调用时的数据类型动态生成模板函数非常神奇，那么它背后到底是如何运作的呢？

实际上，我们可以把“`typename` 标识符”看成是函数模板的参数，不同参数可以产生不同版本的函数。编译器在编译 `mymax()` 函数调用时，可以根据参数的类型推导出函数模板的这个类型参数。例如，代码中第一次调用 `mymax()` 函数，其参数都是整数，则编译器推导出

其模板参数 T 为 int, 此时, 编译器就会以函数模板为样板, 也就是用实际的数据类型替换函数模板中的类型参数 T, 自动为这个函数调用生成一个整型数的版本。

```
// 整型数版本的 mymax() 模板函数
int mymax( int a, int b )
{
    return a > b ? a : b ;
}
```

当主函数以整型数为参数调用 mymax() 函数时, 实际上执行的是上面生成的整型数版本的 mymax() 函数。同理, 当以浮点数为参数调用 mymax() 函数时, 编译器也会为这个 mymax() 函数调用生成一个浮点数版本的 mymax() 函数。函数模板生成重载函数如图 8-2 所示。

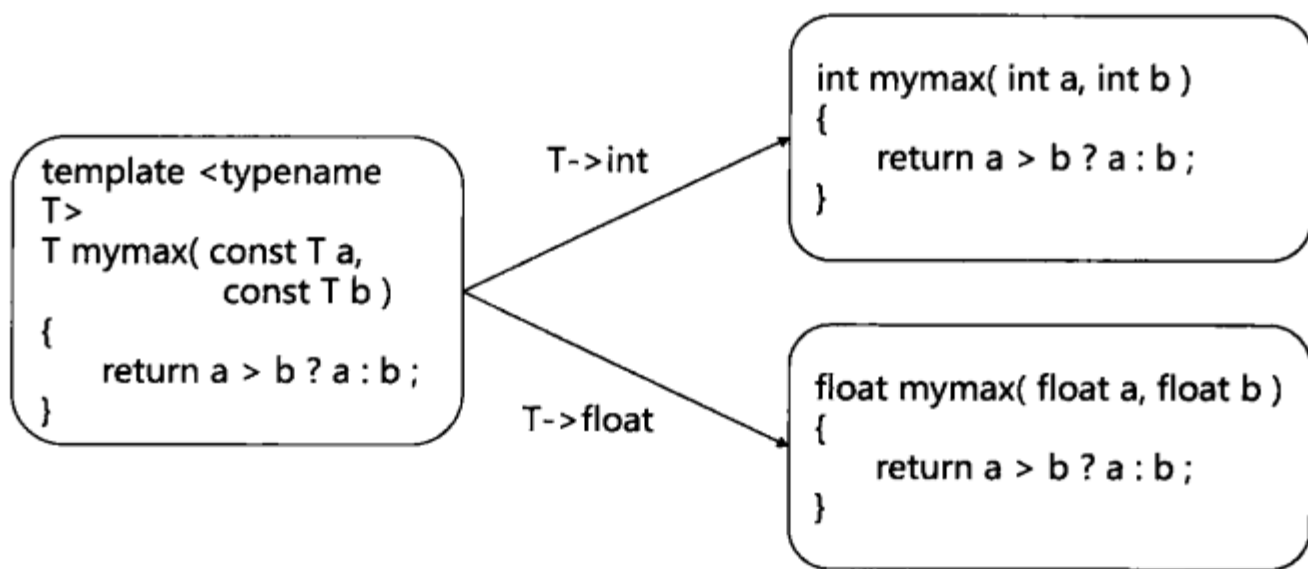


图 8-2 函数模板生成重载函数

在调用模板函数的时候, 编译器会根据调用模板函数的参数类型自动推导出模板函数的类型参数。但在某些特殊情况下, 例如, 模板参数 T 没有出现在函数的参数列表中, 编译器就无法根据调用模板函数时的参数推导出模板函数的类型参数, 或者编译器推导出的是一个我们并不期望的错误类型参数。这时, 就需要在模板函数后加上 “<>” 指明其类型参数, 显式地调用模板函数的某个特定版本。例如:

```
// 显式地指明 mymax() 模板函数的类型参数是 string 类型,  
// 它将调用 mymax() 函数的 string 类型版本  
string strMax = mymax<string>("Chen", "Jia");  
cout<<"较大的字符串是"<<strMax<<endl;
```

从上面代码的输出结果中可以发现, 在 “Chen” 和 “Jia” 这两个字符串中, 较大的字符串是 “Jia”。但是实际上, 我们期望的结果是长度较长的 “Chen” 这个字符串。虽然模板函数的意义是为不同的数据类型提供通用的算法, 但是有的时候, 模板函数需要对特定的数据类型进行特殊处理。例如, mymax() 函数会使用 “>” 运算符比较两个参数的大小并返回其中较大的一个。在大多数情况下, 这种比较都是合理的。但是, 当参数是 string 类型时, 它会逐个比较字符串中的字符并返回字符较大的字符串, 这也就是为什么 “Jia” 字符

串会大于“Chen”字符串的原因。但实际上，我们希望 mymax()函数返回的是两个字符串中字符串长度较长的字符串。在这种情况下，就需要对模板函数进行特化，实现特定类型的模板函数。

```
// 利用模板特化，实现特定类型的 string 的模板函数
template <>
string mymax<string>( const string a, const string b )
{
    return a.length() > b.length() ? a : b ;
}
```

有了某个特定类型的模板特化之后，当使用这一类型的参数调用模板函数时，编译器将使用模板特化后的模板函数，而如果是其他类型的参数，仍将使用模板函数的普通版本。例如：

```
// 默认情况下，调用 mymax<char*>版本，
// 比较字符串的字符，较大的字符串是“Jia”
string strMax = mymax("Chen", "Jia");
cout<<"使用普通版本，较大的字符串是"<<strMax<<endl;

// 显式指明模板参数类型，调用模板特化后的 mymax<string>版本，
// 比较字符串的长度，较长的字符串是“Chen”
strMax = mymax<string>("Chen", "Jia");
cout<<"使用模板特化版本，较长的字符串是"<<strMax<<endl;
```

8.2.2 类模板

函数模板代表了一类函数，它可以以一致的算法流程来处理各种不同类型的数据。但是，函数模板只是一个算法处理过程，它并不能维护算法过程中的数据。例如，一个算法不可能知道它上一次调用时的数据。为了解决这个问题，可以将算法组织到一个算法类中，利用类的成员变量来维护算法过程中的数据，利用类的成员函数来代表算法。跟函数模板相同，在C++中也可以将类模板化，利用类模板来实现对不同类型数据的统一处理。

类模板的声明形式跟函数模板类似，其语法格式如下：

```
template <typename 标识符>
class 类名
{
    // 类的定义
}
```

其中，template 是声明类模板的关键字，它表示接下来的语句将声明一个模板。typename 所定义的标识符实际上就是类模板的参数，模板参数可以是一个，也可以是多个。typename 关键字也可以使用 class 关键字代替。所有模板参数都应该是抽象化的结果，不应是具体的数据类型，例如 int 或者 float 等。因为类模板只是定义一个模板，其中的数据类型并没有确定，

所以类模板中的数据类型都应该使用抽象的数据类型，无论是类的成员函数的参数或返回类型，还是类的成员变量，前面都要加上形参类型。例如：

```
// 定义一个用于比较两个数的类模板
template <typename T>
class compare
{
public:
    // 构造函数，实际上它相当于一个函数模板
    compare(T a, T b)
        :m_a(a),m_b(b)
    {}

    // 比较类的接口函数
    // 类模板中的函数都类似于函数模板
public:
    // 返回两个数中的较小值
    T min()
    {
        return m_a > m_b ? m_b : m_a;
    }
    // 返回两个数中的较大值
    T max()
    {
        return m_a > m_b ? m_a : m_b;
    }
    // 类模板的成员变量，其类型将根据类模板
    // 实例化时的具体类型而定
private:
    T m_a;
    T m_b;
};
```

在这段代码中，定义了一个用于比较两个数据大小的类模板 `compare<T>`。这个类模板有两个成员变量，分别用于保存比较的两个数，同时它提供了两个成员函数 `max()` 和 `min()`，分别返回两个数中的较大值和较小值。有了这个类模板，就可以方便地用它来对各种类型的数据进行比较，例如：

```
int _tmain(int argc, _TCHAR* argv[])
{
    // 用 int 数据类型对类模板进行实例化
    // 比较两个整数的大小
    compare<int> intcompare(2,3);
    wcout<<intcompare.max()<<"大于"<<intcompare.min()<<endl;

    // 用 string 数据类型对类模板进行实例化
    // 比较两个字符串的大小
    compare<string> stringcompare("A","B");
    cout<<stringcompare.max()<<"大于"<<stringcompare.min()<<endl;
```

```
    return 0;
}
```

在主函数中，首先使用 `int` 数据类型实例化了一个 `compare<T>` 类模板，形成了一个新的模板类 `compare<int>`；然后定义了一个 `compare<int>` 类的对象 `intcompare`，用于比较两个整数。同样，使用 `string` 数据类型实例化 `compare<T>` 类模板可以得到模板类 `compare<string>`，这个类可以用于比较两个字符串的大小。从类模板到模板类实例的过程如图 8-3 所示。

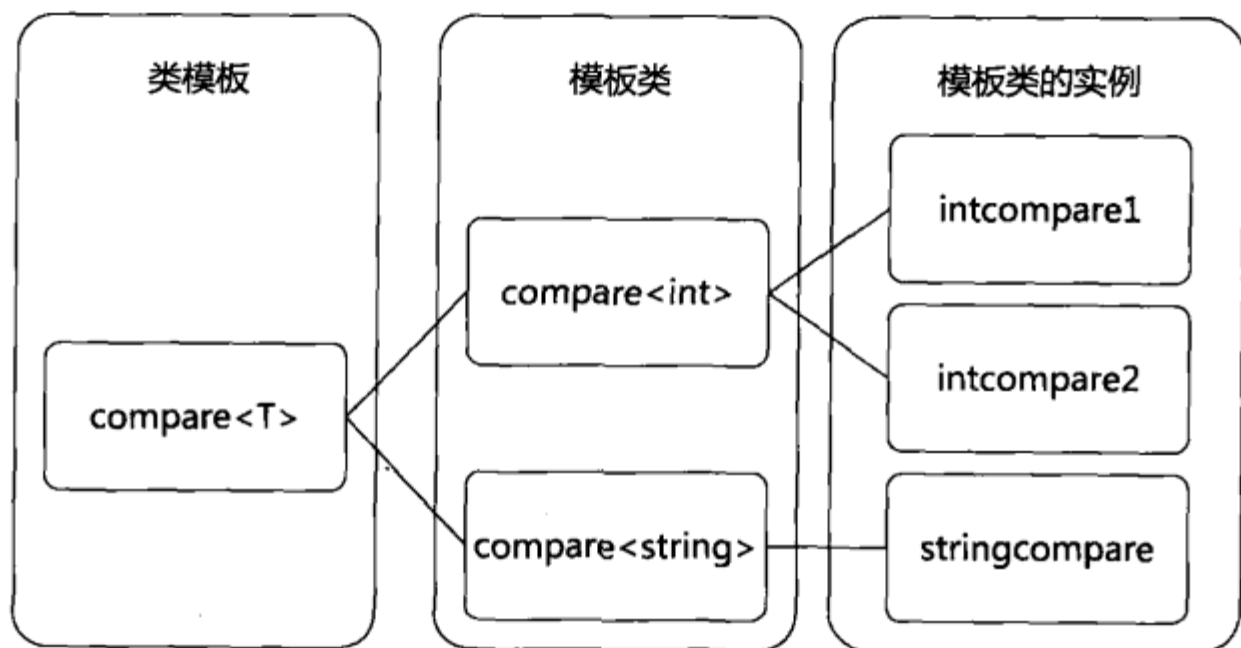


图 8-3 从类模板到模板类实例的过程

从图 8-3 中可以看到，类模板实际上是多个功能相同的类的一种抽象，它可以以数据类型为参数，根据不同的数据类型实例化成不同的具体模板类，从而处理不同类型的数据。

8.2.3 模板的实例化

所谓模板的实例化，就是根据使用模板时给定的具体数据类型，生成采用特定模板参数组合的具体类或函数（实例）的过程。例如，程序中定义了一个类模板 `compare<T>`，在使用这个类模板的时候，需要给定这个类模板的模板参数，也就是具体的数据类型。编译器会根据具体的模板参数生成相应的模板类。比如，如果给定 `int` 作为类模板的参数，就会生成一个模板类 `compare<int>`；同样，如果以 `string` 作为类模板的参数，就会生成模板类 `compare<string>`。通过使用具体的数据类型替换类模板定义中的模板参数，可以定义这些新的类，生成具体的实例源代码。换句话说，模板的实例化也就是在工作总结模板上填上自己的名字，形成自己的工作总结的过程。在 C++ 世界中，模板有以下两种实例化的方式。

1. 隐式实例化

当使用模板函数或模板类时，是需要这些函数或者类的实例的。编译时，如果这些实例不存在，则编译器会自动地隐式实例化模板参数组合的模板，在当前代码单元之前插入模板

的实例化代码。例如，当创建模板类的对象时，编译器会进行模板类的隐式实例化。

```
// 隐式实例化模板类 compare<int>
compare<int> intcompare(2,3);
```

2. 显式实例化

编译器仅为实际使用的那些模板参数组合而隐式实例化模板。同时，C++世界也提供了显式实例化模板的功能。例如，要显式实例化模板函数，需要在 `template` 关键字后接函数的声明（不是定义），且函数标识符后接模板参数。

```
// 显式实例化 int 版本的 mymax 模板函数
template int mymax<int>( int, int );
```

当编译器推断出模板参数时，模板参数可以省略。例如：

```
// 这里省略模板的实际参数<int>，由编译器自动推断模板函数的模板参数
template int mymax( int, int );
```

要显式实例化模板类，可以在 `template` 关键字后接类的声明（不是定义），且在类标识符后接模板参数。例如：

```
// 显式实例化模板类 compare<int>;
template class compare<int>;
```

显式实例化类时，所有的类成员也必须实例化。

使用显式实例化，可以控制模板实例化发生的时间。但是，当一个类模板被显式实例化时，即使并没有用到这些成员函数，它的所有成员函数也将实例化。所以，在大多数情况下都采用隐式实例化，只有在必要的时候，例如类模板的声明和定义放置在不同的.h 文件和.cpp 文件中时，才采用显式实例化。

8.2.4 用模板实现通用算法

不用羡慕 STL 提供了很多通用算法、可以处理各种数据类型。有了模板的帮助，就可以通过模板函数或者模板类创建自己的通用算法，甚至可以实现自己的 STL。正所谓自己动手，丰衣足食。下面以一个实际的例子来介绍如何使用模板实现通用算法。

在软件设计中，通常有一个撤销（undo）和恢复（redo）的通用功能。撤销和恢复可以是一个输入的数据，也可以是一个自定义的动作。面对这种需要处理多种数据类型的情况，可以使用模板参数来表示具体的数据类型，在类模板或者函数模板中操作模板参数来代替对具体数据类型的操作，从而可以让这些类和函数能够处理各种数据类型，应对各种需要实现的通用算法。在这里，可以利用模板实现一个通用的动作容器，这个容器可以容纳各种类型的数据，并且这个容器还可以对其中的数据进行操作。

```

// 动作容器模板类
// 使用 T 作为模板参数
template <class T>
class actioncontainer
{
public:
    // 构造函数，初始化动作的位置
    actioncontainer()
    {
        m_nRedoPos = 0;
        m_nUndoPos = 0;
    }

    // 容器的接口函数
    void add(T value);           // 向容器中添加新的动作
    T redo();                   // 恢复上一步动作
    T undo();                   // 撤销上一步动作

// 容器的属性
private:
    // 记录动作的位置
    int m_nRedoPos;
    int m_nUndoPos;
    // 使用常量表示可以撤销的动作数，
    // 也就是容器的容量
    const static int ACTION_SIZE = 5;
    // 使用数组记录恢复和撤销的动作
    // 这里数组元素的类型也是模板参数，
    // 它在类模板实例化的时候才确定
    T m_RedoAction[ACTION_SIZE];
    T m_UndoAction[ACTION_SIZE];
};

// 模板类的成员函数实现
// 向容器中添加动作。这里的 T，可以是一个基本数据类型，
// 也可以是一个自定义类型
template <class T>
void actioncontainer<T> ::add(T value)
{
    // 判断容器中的动作数目是否超过容器的容量
    if( m_nUndoPos >= ACTION_SIZE )
    {
        // 如果容器已满，则调整添加的位置
        // 将新动作添加到容器末尾
        m_nUndoPos = ACTION_SIZE - 1;
        // 将容器中的已有动作前移一个位置
        for(int i = 0; i < ACTION_SIZE; ++i)
            m_UndoAction[i] = m_UndoAction[ i + 1 ];
    }
    // 将新动作添加到容器中
    m_UndoAction[m_nUndoPos++] = value;
}

```

```

}

// 撤销上一步动作
template <class T>
T actioncontainer<T>::undo()
{
    // 将撤销的动作复制到恢复数组中
    m_RedoAction[m_nRedoPos++] = m_UndoAction[--m_nUndoPos];

    // 返回撤销的动作
    return m_UndoAction[m_nUndoPos];
}

// 恢复上一步动作
template <class T>
T actioncontainer<T>::redo()
{
    // 将恢复的动作复制到撤销数组中
    m_UndoAction[m_nUndoPos++] = m_RedoAction[--m_nRedoPos];

    // 返回恢复的动作
    return m_RedoAction[m_nRedoPos];
}

```

在这里，定义了一个类模板 `actioncontainer<T>`，这个类可以容纳动作并且能够进行撤销和恢复操作。在这个模板类的内部，使用了两个数组来分别记录撤销和恢复的动作。这里的动作，实际上是广义上的一种数据类型，它可以是一个基本数据类型，比如 `int` 或者 `float`，也可以是自己定义的数据类型，这样就使得这个动作容器具有广泛的通用性，可以处理各种数据类型。在这个动作容器类模板 `actioncontainer<T>` 中，还定义了它的成员函数，用于操作这个容器中的数据，其中包括添加新的动作到容器中、撤销或者恢复动作等。当然，这里成员函数同样是对抽象的模板参数进行操作，其中并不涉及具体的数据类型，也就是说，这些操作与具体的数据类型无关。

完成容器的定义后，就可以使用这个新的容器来处理各种数据了。

```

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个 int 类型的动作容器
    // 这样这个动作容器就可以容纳 int 类型的数据
    actioncontainer<int> intaction;

    // 向容器中添加新的动作，也就是整数
    intaction.add(1);
    intaction.add(2);
    intaction.add(3);
    intaction.add(4);

    // 撤销上一步动作
    // 这时，nUndo 的值为 4
}

```

```
int nUndo = intaction.undo();  
// 再次撤销上一步动作, nUndo 的值为 3  
nUndo = intaction.undo();  
// 恢复上一步动作, nRedo 的值为 3  
int nRedo = intaction.redo();  
// 再次恢复上一步动作, nRedo 的值为 4  
nRedo = intaction.redo();  
  
return 0;  
}
```

在主函数中, 首先使用 `int` 类型实例化了一个容器模板类 `actioncontainer<int>`, 并且创建了一个模板类的实例对象, 这表示这个容器装的是 `int` 类型的数据。接着向动作容器中添加新的动作。这里的动作, 也就是整数数据。最后, 通过这个容器提供的函数, 就可以对容器进行操作, 模拟用户的撤销和恢复操作, 容器会返回相应的动作, 也就是我们添加进去的整数数据。在实际应用中, 可以使用自定义的数据类型作为动作来实现更加复杂的功能。

通过简单的动作容器模板类, 就可以实现通用的撤销和恢复功能, 这正体现了模板的威力所在。这也可以让程序员一劳永逸, 用一个模板通吃各种数据类型, 再也不用为了处理多种数据类型而去创建同一个函数或者类的多个版本。

知道更多: 什么是泛型编程

通过使用模板, 可以使程序具有更好的代码重用性。泛型编程 (generic programming) 就是一种大量应用模板来实现更好的代码重用性的编程方式。

跟面向对象编程不同, 泛型编程并不要求我们通过额外的间接层来调用函数, 它可以编写完全一般化并可重复使用的算法, 其效率与针对某特定数据类型而设计的算法相同。所谓泛型 (genericity), 是指具有在多种数据类型上皆可操作的含义。它允许程序员在强类型程序设计语言中编写代码时定义一些可变部分, 这些可变部分在使用前必须指明。在 C++ 世界中, 泛型是通过模板机制来实现的。而泛型编程, 主要是通过模板机制来构建一类操作类似但数据类型不同的程序, 其中就包括大量的函数模板和类模板。

泛型编程的代表作品 STL 是一种高效、泛型、可交互操作的软件组件。STL 以迭代器和容器为基础, 是一种泛型算法 (generic algorithms) 库, 容器的存在使这些算法有东西可以操作。STL 巨大、可以扩充, 它包含很多基本算法和数据结构, 而且将算法与数据结构完全分离, 其中算法是泛型的, 不与任何特定的数据结构或对象类型联系在一起。

STL 中的容器管理数据

虽然 STL 中有算法、容器和迭代器三大核心组件，但是最受 C++ 程序员欢迎的还是“大肚能容”的容器。

程序就是用来处理数据的。一个程序从它开始执行到执行完毕，无时无刻不在跟各种数据打交道。在一个程序中，少量的数据的处理使用单个变量就可以完成。但如果要处理大量的数据，就需要对这些数据进行良好的管理。正所谓忆苦才能思甜，在容器诞生之前，程序员要使用数组保存和管理大量数据。使用数组，程序员需要自己做很多事情，比如管理内存、维护数组中保存的数据、防止数组访问越界等。随着 STL 容器的到来，程序员纷纷扔掉镰刀、锄头，开上了既省时又省力的拖拉机。相比于数组，容器更加强大而灵活：它们的容量可以动态地扩充和缩减；它们可以自己管理内存；它们可以记住自己包含了多少数据元素；它们限制了自己所支持的操作复杂性；它们还可以……

关于容器的好处说不完，这也就不难理解它们为何如此受到程序员的欢迎了。STL 容器不是一般地好，而是确实很好。

9.1 容器就是 STL 中的瓶瓶罐罐

容器，简单来讲，就是能够保存其他类型的对象的类。它是 STL 的关键部件之一，用于管理算法要处理的大量特定类型的数据。

STL 中的容器分为顺序容器（sequence container）和关联容器（associative container）两种，如图 9-1 所示。

1. 顺序容器

顺序容器将对象组织成有限线性集合，所有对象都是同一类型的。STL 中包括三种基本顺序容器：向量（vector）、线性表（list）和双向队列（deque）。基于这三种基本顺序容器，又可以构造出一些专门的容器，用于比较特殊的数据结构，包括堆（heap）、栈（stack）、队列（queue）及优先队列（priority queue）。

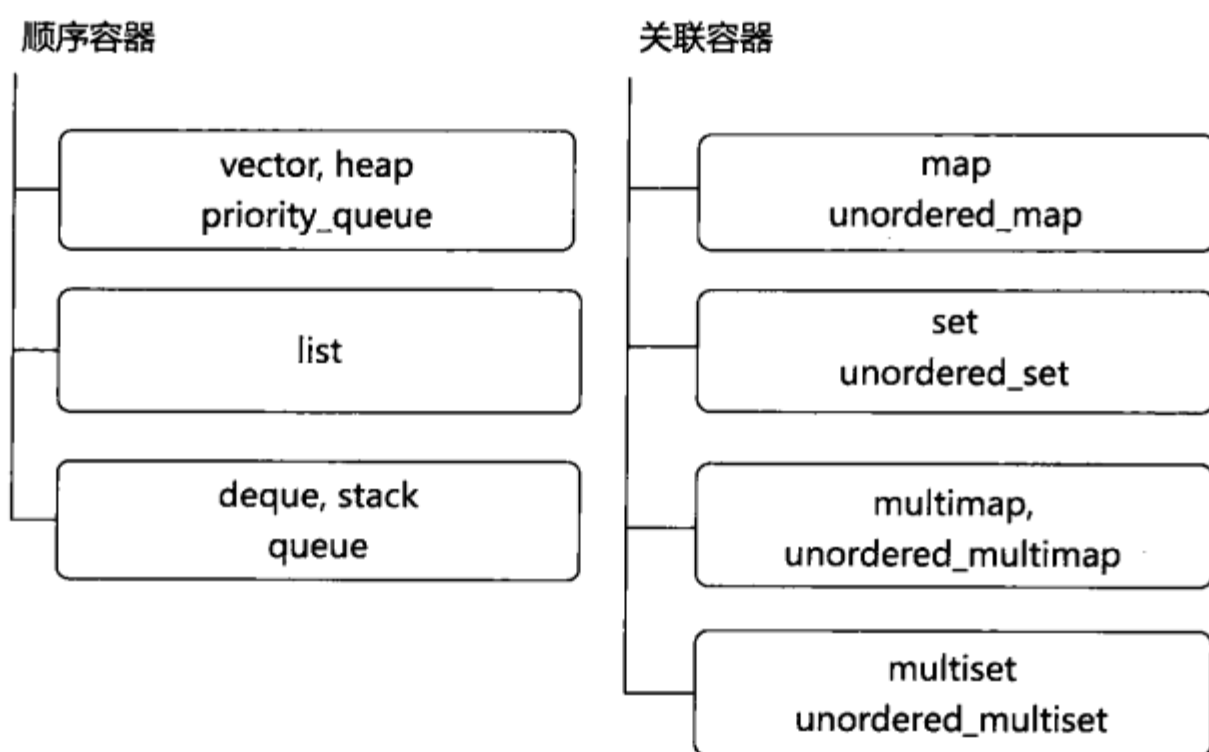


图 9-1 STL 中的容器

2. 关联容器

关联容器所容纳的对象是由{键, 值}对组成的, 它提供了基于键/值的数据快速检索能力。容器中的元素在加入容器时就已经被排好序, 检索数据时可以按照二分搜索提高检索的效率。STL 中有 8 种关联容器。当一个键对应一个值时, 可以使用集合 (set) 和映射 (map) 存放这种一一对应的数据。若同一个键对应多个值时, 则可以使用多集合 (multiset) 和多映射 (multimap) 存放这种一对多的数据。同时, 集合和映射又可以根据内部实现机制的不同, 分为基于红黑树实现的 set、multiset、map 和 multimap, 以及基于哈希表实现的 unordered_set、unordered_multiset、unordered_map 和 unordered_multimap。使用时可以根据需要灵活地进行选择。

STL 中的容器, 实际上就是一些数据结构的模板类。当使用这些容器时, 需要根据它们容纳的数据类型对其进行实例化, 产生相应的容器模板类。利用这些实例化之后的模板类, 才能创建自己的容器对象实例。例如, 创建一个可以容纳 Employee 自定义数据结构的 vector 容器。

```
// 可以容纳 Employee 结构的 vector 容器  
vector<Employee> vecEmployee;
```

在这里, 首先以 Employee 这种数据类型作为模板类的参数实例化产生一个模板类 vector<Employee>, 它表示 vector 容器可以容纳 Employee 这种数据类型的数据对象。有了容器模板类, 就可以创建真正的实体对象 vecEmployee, 而这个对象就是我们真正使用的容器对象。

有了容器对象, 接下来的任务就是向容器中添加数据, 把数据装到容器中进行管理。

9.1.1 操作容器中的数据元素

在 STL 中，容器都提供了自己的函数来操作容器中的数据。利用这些函数，操作容器中的数据将更加方便：将数据元素添加到容器中，或者删除容器中的数据元素等。例如，可以使用 vector 容器的 `push_back()` 函数将数据元素添加到 vector 容器中。

```
// 定义一个可以保存 int 类型数据的容器
vector<int> vecSalary;
// 接收用户输入并将数据保存到容器中
int nInput = 0;
do
{
    cin>>nInput;
    if ( 0 == nInput )
        break;
    // 通过 push_back() 函数将数据装入容器中
    vecSalary.push_back( nInput );
} while( true );
```

在这段程序中，首先定义了一个可以容纳 int 类型数据的 vector 容器 `vecSalary`，然后标准输入流对象 `cin` 会接收用户的输入并将其保存到变量 `nInput` 中，接着通过 `push_back()` 函数将变量 `nInput` 装入容器 `vecSalary` 中。这样，就实现了将数据装入容器的操作。

除了数据的装入操作之外，大多数容器都提供了针对容器中的数据元素的常用操作函数，比如元素的删除、插入、交换和清空等。如果想操作容器中的元素，则只需要调用相应的操作函数就可以了。例如：

```
// 向 vector 容器的开始位置插入一个数据
vecSalary.insert( vecSalary.begin(), 4999 );
// 删除 vector 容器中的前三个数据
vecSalary.erase( vecSalary.begin(),
                 vecSalary.begin() + 3 );
// 清空 vector 容器中的所有数据
vecSalary.empty();
```

9.1.2 使用迭代器访问容器中的数据元素

容器提供的各种操作函数，可以方便将数据装入容器，同时对容器中的数据进行管理。但是，并不是将数据装入容器就完事了，更多情况下，还要访问容器中的数据供算法使用。为了访问容器中的数据，可以将容器和算法结合起来，STL 提供了迭代器这个黏合剂。

迭代器提供了一个对容器中数据元素进行访问的方法。我们先将迭代器指向容器中的某个数据元素，然后通过这个迭代器访问它所指向的数据元素。从表现上来看，迭代器如同一个指针，它指向容器中的各个数据元素，并且可以通过它访问所指向的数据元素。从这个意义上讲，C++ 语言中的指针也可以看成是一种迭代器，但是，迭代器不仅仅是指针，因此不

能认为它们一定具有地址值。例如：

```
// 定义一个 vector<int>容器的迭代器
vector<int>::iterator it;
// 将迭代器 it 指向 vector 容器的起始位置，这时 it 指向的是 vector 容器中的第一个元素
// vector 容器的 begin() 函数返回的是指向其起始位置的游标 (iterator)
it = vecSalary.begin();

// 通过迭代器访问容器中的数据元素
// 跟指针类似，在迭代器前使用 “*” 运算符就可以得到它所指向的数据元素
// 如果工资小于 2000 元，则增加为原来的 120%
if( *it < 2000 )
{
    // 通过迭代器读/写它所指向的数据元素
    *it = (*it) * 1.2;
}
```

除了可以使用迭代器访问容器中的单个数据元素之外，还可以使用两个迭代器定义容器中数据元素的某个范围。例如，可以使用一对迭代器指定一个容器中的前 4 个元素这样一个范围。例如：

```
// 定义一个 vector<int>容器的迭代器，表示起始位置
vector<int>::iterator itfrom;
// 将迭代器指向 vector 容器的起始位置 itfrom = vecSalary.begin();
// 定义一个 vector<int>容器的迭代器，表示终止位置
vector<int>::iterator itto;
// 将表示终止位置的迭代器指向 vector 容器起始位置后的第 4 个数据元素
itto = vecSalary.begin() + 3;
```

在这段代码中，定义了两个 vector 容器的迭代器，分别指向 vector 容器中的第一个元素和第三个元素，以此来表示一个数据元素的范围，如图 9-2 所示。

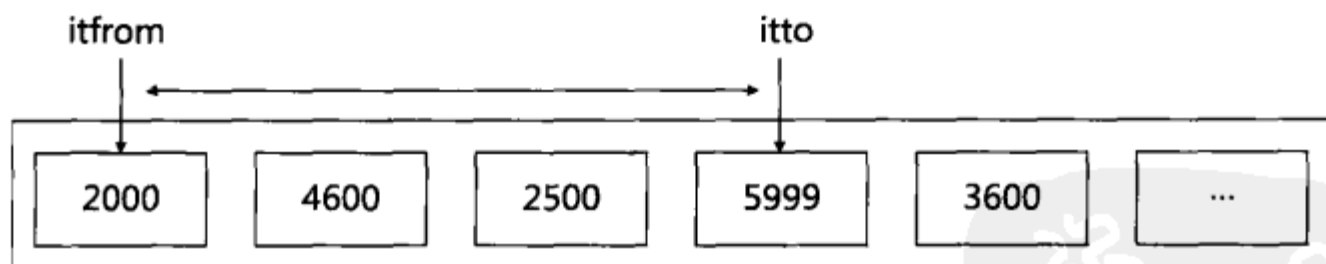


图 9-2 使用迭代器表示容器中的范围

按照使用的目的不同，各个容器都提供了多种类型的迭代器。例如，有的迭代器只能对数据进行只读访问；有的迭代器只能向前移动等。通常，我们把迭代器看成是可以访问容器中元素的一种对象，把它作为一个变量来创建。但是在使用上，迭代器更像一个指针，跟指针相似，可以在迭代器对象前加上 “*” 运算符来获取这个迭代器所指向的数据。另外，还可以使用自增运算符 “++” 或者自减运算符 “--” 来将迭代器向前或者向后移动一个位置，让它指向容器中的其他位置的数据元素。如果迭代器到达容器中的最后一个元素的后面，则迭

代器变成 past-the-end 值。使用 past-the-end 值的迭代器来访问数据元素是非法的，就好像使用 NULL 值的指针一样。例如：

```
// 使用迭代器循环遍历容器中的数据
for( vector<int>::iterator it = vecSalary.begin();
// 将迭代器指向容器的起始位置
    it != vecSalary.end();    // 判断是否到达容器的最后位置
    ++it )    // 通过自增运算符将迭代器指向容器中的下一个元素
{
    // 通过迭代器访问它所指向的数据元素
    nTotal += (*it);
}
```

在这段代码中，使用了一个迭代器来循环遍历容器中的所有数据。在 for 循环中，首先定义了一个迭代器并将它指向了容器的起始位置；在循环体中，通过迭代器访问它所指向的数据元素，每次循环之后，都通过自增运算符将迭代器指向容器的下一个元素，直到迭代器到达容器的最后一个元素为止。这里需要注意的是，当判断迭代器是否到达循环范围的结束位置时，并没有使用通常意义上的“<”运算符来判断当前迭代器是否小于结束位置，这是因为“<”运算符在某些容器中没有定义，为了一致性，使用“!=”运算符来判断迭代器是否到达循环的结束位置。

9.1.3 容器的使用说明书

容器虽好，也得小心轻放、注意容器的各种使用规则。这里有一份容器的使用说明书，只有按照说明书上的使用规则，才能将容器用得恰到好处，发挥它应有的作用。这份容器使用说明书是这样写的：

- 容器可以用来存放对象或者对象的指针。一般来说，容器不太关心里面存放的是什么东西，既可以在容器中存放普通的数据，也可以存放类的实例对象，或者指向这些实例对象的指针。在创建容器时，可以通过容器模板类的参数指定这个容器将要存放的内容。例如：

```
// 用于存放整型数的 vector 容器
vector<int> vecnSalary;
// 用于存放字符串的 list 容器
list<string> listName;
// 用于存放 Employee* 指针的 map 容器
map<int, Employee*> mapEmployee;
```

一般来说，既可以在容器中存放对象，也可以存放这些对象的指针，那么到底该如何选择呢？如果使用的是基于连续内存的容器，例如 vector 容器等，当在这些容器中插入或者删除元素时，往往会引起内存的重新分配或者内存的复制移动。在这种情况下，为了提高内存操作的性能，我们优先选择保存对象的指针，因为指针的体积通常比对象的更小。对基于节

点内存的容器，当进行数据元素的操作时则很少有内存的复制移动，所以在这种容器中保存对象或者指向对象的指针并无性能上的差别。但是从方便使用的角度考虑，可以优先选择保存对象。如果需要保存一些机器资源（例如，文件句柄、命名管道、套接字或者其他类似的资源），那么通常选择保存指向这些对象的指针。

- 小心清理容器中保存的对象指针。如果容器中保存的是对象，那么在容器析构的时候容器会自动清理这些对象。但是，如果容器中保存的是对象的指针，那么这些指针的清理工作就是程序员的责任了。我们应该在容器使用完毕后，注意清理其中保存的指针，释放这些指针所指向的对象。例如：

```
// 创建一个存放 Employee* 指针的 vector 容器
vector<Employee*> vecEmployee;
// 对容器进行操作...

// 在容器使用完毕后，清空容器中保存的指针，
// 释放这些指针所指向的对象
for( auto it = vecEmployee.begin();
    it != vecEmployee.end(); ++it )
{
    // 判断指针是否为 NULL，
    // 如果不为 NULL，则释放指针指向的对象
    if( NULL != (*it) )
        delete (*it);    // 释放指针指向的对象

    (*it) = NULL;        // 将指针设置为 NULL，防止误用
}
// 清空整个容器
vecEmployee.clear();
```

这里通过循环遍历容器中保存的每个指针，释放这些指针所指向的对象，将整个容器清空，最终完成容器的手动清理工作。

- 为容器中的对象实现拷贝构造函数和赋值操作符。如果需要将某个对象保存到容器中，实际上 STL 会重新生成一个此对象的拷贝，然后将这个拷贝保存在容器中，源对象将不再使用。所以，为了保证将对象装入容器时对象能够被正确地拷贝，需要实现这个对象的类的拷贝构造函数和赋值操作符。例如：

```
// 将会保存到容器中的员工类
class Employee
{
public:
    // 实现拷贝构造函数
    Employee(const Employee& rSource )
    {
        // 利用赋值操作符实现对象的拷贝
        *this = rSource;
    }
};
```



```

};
// 实现赋值操作符，这里对类的每个属性都进行了合理的初始化
Employee& operator = (const Employee& rSource )
{
    m_nSalary = rSource.m_nSalary;
    m_strName = rSource.m_strName;

    return *this;
};
// 类的属性
private:
    int m_nSalary;
    string m_strName;
};

```

当然，很多时候无须自己实现类的拷贝构造函数，编译器会自动创建一个拷贝构造函数，以拷贝内存的方式完成类对象的拷贝。但是，这里要特别注意的是，内存拷贝并不能保证类对象被正确地拷贝，特别是当类中有指针作为成员属性时，需要谨慎地实现自己的拷贝构造函数，以保证类对象在装入容器时能够正确地拷贝。

- 使用迭代器删除容器中的数据元素需谨慎。当使用迭代器删除容器中的数据元素时，容器中的元素位置将发生变化，所以迭代器所代表的当前位置也会发生变化，这一点需要特别注意。例如，要删除一个容器中大于 1000 的数，按照思维习惯，我们会遍历容器中的数据元素，遇到符合条件的元素就进行删除，实现如下：

```

// 循环遍历删除容器中的元素
for( auto it = vecSalary.begin();
    it != vecSalary.end(); ++it )
{
    // 遇到符合条件的元素就进行删除
    if( *it > 1000 )
        vecSalary.erase(it);
}

```

可是，当实际运行这段代码的时候，发现其运行结果并非想象的那样。这是因为在删除过程中，在删除某个元素后，vector 容器后面的元素会自动向前移动一个位置，以保持 vector 容器内存的连续性，这时的迭代器实际上指向的是被删除元素后的第一个元素。当进入下次循环的时候，迭代器向后移动一个位置，实际上指向的是被删除元素后的第二个元素，中间跳过了一个元素，这就很可能造成漏掉某些元素的检查而导致删除不完全。所以，需要在每次删除动作发生后，重新设置当前迭代器的值，将它指向正确的容器位置。

```

// 循环遍历删除容器中的元素
for( auto it = vecSalary.begin();
    it != vecSalary.end(); )
{
    // 遇到符合条件的元素就进行删除
    if( *it > 1000 )
        it = vecSalary.erase( it ); // 删除完成后，将迭代器重新指向正确的位置
}

```



```
        else
            ++it;                // 如果不删除当前元素，则将迭代器指向元素的下一个位置
    }
```

熟读容器的使用说明书，了解容器的各种使用规则和注意事项，可以让容器使用起来更加舒心、更加得心应手。

9.1.4 如何选择合适的容器

STL 提供了各种各样的容器，这些容器各有所长，它们可以应用在各种不同的场景下，我们可以根据自己的需要选择合适的容器，这样才能发挥容器最大的效果。那么，面对如此多的容器，我们又该如何选择呢？

《圣经》上说：“你应该了解真相，真相会使你自由。”所以，只有了解各种容器的本质，才能做出正确的选择。

在 STL 中，按照内存组织的不同，容器分为顺序容器和关联容器。其中，顺序容器就是连续内存容器，也叫基于数组的容器，是在一个或多个内存块（动态分配的）中保存它们的元素。如果一个新元素被插入或者已存在的元素被删除，其他在同一个内存块的元素就必须向上或者向下移动来为新元素提供空间，或者填充原来被删除的元素所占用的空间。这种移动会影响容器的效率和异常安全。STL 所提供的连续内存容器有 `vector`、`array` 和 `deque` 等。`vector` 容器在内存组织形式上类似于数组，但是它也提供了很多额外的功能，利用 `vector` 容器，可以比数组更方便地存储和管理数据。例如，可以通过具有边界检查功能的 `at()` 函数访问 `vector` 容器中的数据；通过 `push_back()` 或 `erase()` 函数进行数据的添加和删除，以及自动的内存管理等。`vector` 容器的这些优势，使我们在需要保存大量数据的时候，优先选择 `vector` 容器。

除了连续内存容器之外，STL 中还有基于节点的容器，也就是关联容器。这种容器在每个内存块（动态分配的）中只保存一个元素。容器元素的插入或删除只影响指向节点的指针，而不是节点自己的内容。所以当有元素插入或删除时，元素值无须移动。表现为链表的容器，例如 `list` 容器，是基于节点的容器，同时 STL 中其他所有的标准关联容器，例如 `set` 和 `map` 等，都是基于节点的容器。因为 `map` 容器能够保存的都是具有某种关系的{键，值}数据对，所以 `map` 容器常常被用来实现查找表，或者用来存储稀疏数组或稀疏矩阵。

连续内存容器和基于节点的容器提供给了程序员不同的复杂度，因此可以这样来进行选择：`vector` 是一种可以默认使用的序列的类型，大多数情况下我们都会选择使用 `vector` 容器；但是当需要很频繁地对序列的中部进行插入和删除时应该使用 `list` 等基于节点的容器；当大部分插入和删除发生在序列的头部或尾部时可以选择 `deque` 容器。

9.2 vector 容器是数组的最佳替代者

在没有 STL 的时代，如果想在程序中保存大量的数据，则只能选择数组这种“粗鲁”的方式。但是 STL 中的 vector 容器出现后，无论是对内存的管理还是对数据元素的访问，vector 都要优于数组，因此，vector 成为数组的最佳替代者。STL 中的容器虽多，但是 vector 容器是最常用的容器。这正是 STL 中容器三千，我们独爱 vector 一个。

9.2.1 创建并初始化 vector 对象

如果要将大量的对象保存到容器中，使用数组，则可以这样写：

```
// 定义数组的容量
const int MAX_COUNT= 1000;
// 定义保存 Employee 对象的数组
Employee arrEmployee[MAX_COUNT];
```

其中，MAX_COUNT 表示这个数组能够保存的 Employee 对象的最大数量。这种方式有很多的弊端：如果定义的最大数量大于实际的需要，就会造成内存空间的浪费；如果定义的最大数量小于实际的需要，又不利于程序的扩展。同时，对数组元素的访问也非常“粗鲁”，数组无法对索引值进行检查，很容易造成数组的访问越界等内存访问错误；无法在一个数组的中间或者末尾动态地增加数据元素；无法通过传值的方式在函数之间传递一个数组等。为了将程序员从数组的泥潭中拯救出来，STL 中的 vector 容器就可以派上用场了。

vector 容器实际上就是一个动态数组，它是在堆中分配内存的，所有元素连续存放。vector 容器可以根据需要预先分配内存空间，也可以根据实际的需要动态地进行内存的分配。这样就可以做到内存的合理利用。

要使用一个 vector 容器，首先需要创建相应的 vector 容器对象。跟所有容器对象的使用一样，需要先使用容器保存的对象类型实例化一个 vector 容器模板类，然后利用这个模板类创建相应的容器对象实例。

```
#include <vector>           // 引入 vector 所在的头文件
using namespace std;        // 使用 vector 所在的名字空间

// ...
// 先使用 Employee 类实例化 vector 类模板，
// 然后创建实例对象 vecEmployee
vector<Employee> vecEmployee;
```

这样，就得到了一个空的 vector 容器，如图 9-3 所示。如果想得到一个已经保存有默认数据的 vector 容器，则可以在创建容器对象的时候，利用其构造函数指定默认数据及默认数据的个数。当容器对象创建完成时，这些默认数据也就已经保存在容器中了。例如：

```
Employee employee;
// 创建 4 个 employee 对象的副本保存到容器中
vector<Employee> vecEmployee(4, employee);
// 使用 Employee 类的默认构造函数创建 4 个对象保存到容器中
vector<Employee> vecDefEmployee(4);
```

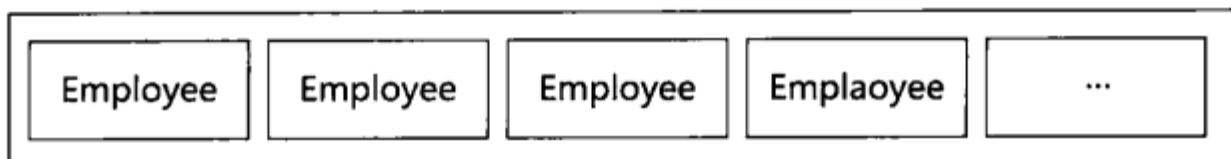


图 9-3 vector 容器

vector 容器的内存是随着元素个数的增加而动态增加的，如果已经预先知道 vector 容器的最大容量，则可以事先使用 reserve() 函数申请足够的内存，为 vector 容器预留相应的元素位置，减少在元素添加过程中内存的动态申请过程。例如：

```
// 为 vector 容器预留 1000 个元素位置
vecEmployee.reserve( 1000 );
```

从上面的例子中可以发现，vector 容器只能保存某种类型的数据。但是有的时候需要同时保存具有某种关系的多组数据。比如，要表述一群人的时候，常常需要保存这群人的姓名、年龄、体重等。在这种情况下，为了避免使用多个 vector 容器来保存多组数据，通常的办法就是利用结构体将多种数据打包成一个新的数据类型，vector 容器再保存这种新的数据类型，以达到同时保存多组数据的目的。例如：

```
// 将表示人的姓名、年龄、体重的数据打包成 Human 结构体
struct Human
{
    string strName;
    unsigned int nAge;
    double fWeight;
};
// 保存 Human 结构体的 vector 容器
// 也就相当于保存了人的姓名、年龄、体重多组数据
vector<Human> vecHuman;
```

虽然这样可以解决一个 vector 容器保存多组数据的问题，但是也需要在代码中定义多个结构体，如果结构体过多，就会使得代码显得庞杂而烦琐。STL 的目的就是让代码优雅而简洁。为了让 vector 容器既能够保存多组数据而又不使代码变得烦琐，STL 为我们引荐了一位打包专家——tuple。

跟结构体可以打包多个数据相似，tuple 也可以将多个有关联的数据类型打包到一起，从而形成一种新的数据类型。比如，描述一个人，通常有他的姓名、年龄、体重等。利用 tuple，可以直接将与“人”这个概念相关的多个属性打包成一种新的数据类型，再利用这种新的数据类型作为 vector 容器保存的对象，轻松完成上面代码中需要定义新的结构体才能完成的功能。

```

// 引入定义 tuple 的头文件
#include <tuple>
using namespace std;

// tuple<string,unsigned int,double>已经是一种新的数据类型
// 它有三个分数据，分别表示人的姓名（string）、年龄（int）和体重（double）
// 使用这种新的数据类型定义一个变量 huChen
tuple<string,unsigned int,double> huChen;
// 使用 make_tuple() 函数对变量 huChen 赋值
tupleHuman = make_tuple("ChenLiangqiao", 28, 66.3);
// 或者更简单地，利用 typedef 为这种新的数据类型定义一个简短的类型名
typedef tuple<string,unsigned int,double> Human;
// 然后使用新的数据类型名定义变量，利用 tuple 的构造函数为变量赋初始值
Human huJia("Jiawei",23,56.3);
// 使用新的数据类型作为 vector 容器实例化参数，
// 定义一个可以保存这种数据类型的 vector 容器
vector<Human> vecHumans;
// 将数据保存到 vector 容器中
vecHumans.push_back(huChen);
vecHumans.push_back(huJia);

```

从上面的代码可以发现，使用 tuple 后代码就变得更加简洁优雅。如果 vector 容器需要保存一组有关联的多个数据，就可以简单地使用 tuple 将其打包成一种新的数据类型，vector 容器直接保存这种打包后形成的数据类型就可以了。

当然，打包了就要拆包。不仅要数据打包，还要从打包完成后的 tuple 数据组中获取各个分数据对其进行读/写操作。为此，STL 提供了 get() 函数，可以用来访问 tuple 数据组的各个分数据。除此之外，STL 还提供了一个 tie() 函数，就像它的名字一样，它可以将表示分数据的多个变量捆绑在一起，接受 tuple 数据组变量的赋值，各个表示分数据的变量也就具有了 tuple 数据组相应分数据的值。例如：

```

// 获取 tuple 数据组变量 huChen 中的第一个数据姓名
cout<<"姓名: "<<get<0>(huChen)<<endl;
// 将 huChen 中的第二个数据年龄加 1，表示又增长了 1 岁
get<1>(huChen) += 1;
cout<<"年龄: "<<get<1>(huChen)<<endl;
// 获取 huChen 中的第三个数据体重
cout<<"体重: "<<get<2>(huChen)<<endl;
// 定义用于保存各个分数据的变量
string strName;
unsigned int nAge;
double fWeight;
// 获取 huChen 中的各个数据，并保存到各个分数据的变量中
tie(strName,nAge,fWeight) = huChen;

```

有了打包专家 tuple 的帮忙，多组数据可以方便地打包成一组数据了。

替代数组的第二种选择——array 容器

作为数组的最佳替代者，vector 容器有无数的优势，但是在某些情况下，这些优势却变成了劣势。比如，要表示一组固定个数的数据，例如一年的 12 个月、一周的 7 天等。在这种情况下，如果仍旧使用 vector 容器来存放这些固定个数的数据，它的优势将变成劣势。vector 容器的动态长度往往会申请比实际保存的数据更多的空间，这样会浪费容器的空间；vector 容器提供了 push_back() 函数可以动态地向容器中添加数据，如果无意中调用了一个 push_back() 函数，就破坏了这组数据的固定长度。既要利用容器的各种优势，又要避免 vector 容器保存固定个数的数据时的各种劣势，STL 提供了替代数组的第二种选择——array 容器。

跟 vector 容器相似，array 容器同样是一个模板类，只不过它需要两个实例化参数，第一个参数表示这个容器可以保存的数据类型，第二个参数表示这个容器可以保存的数据个数。例如：

```
// 引入定义 array 容器的头文件
#include <array>
using namespace std;
// 定义一个只能保存 12 个 string 数据元素的 array 容器
array<string, 12> arrMonths;
// 对容器中的数据元素进行赋值
arrMonths[0] = "Jan";
arrMonths[1] = "Feb";
// ...
arrMonths[11] = "Dec";
```

同样，可以使用迭代器或者 “[]” 运算符访问 array 容器中的数据。

```
// 输出 array 容器中的所有数据
for(array<string, 12>::iterator it = arrMonths.begin();
    it != arrMonths.end(); ++it )
{
    // 使用迭代器访问 array 容器中的数据
    cout<<*it<<endl;
}
```

概括来讲，array 容器除了无法动态地改变它能够保存的数据的个数之外，在操作上，array 容器与 vector 容器并无太大区别。所以，如果有一组固定个数的数据需要保存，则 array 容器将是不二之选。

9.2.2 vector 容器的操作

对于数组，我们都是通过指针或者它的下标来对数组中的元素进行访问的，这样的操作方式非常容易造成数组访问越界的内存访问错误。作为数组的最佳替代者，当然不会采用这么“粗鲁”的方式来操作容器中的元素。为了更加“优雅”地操作容器，它提供了很多操作函数，使用这些函数，可以灵活而安全地完成对 vector 容器的各种操作。vector 容器提供的常用操作函数如表 9-1 所示。

表 9-1 vector 容器的常用操作函数

函 数	说 明
v.empty()	判断容器 v 是否为空。如果 v 为空，则返回 true；否则返回 false
v.size()	返回容器 v 中元素的个数
v.push_back(t)	在容器 v 的末尾增加一个值为 t 的元素
v.pop_back()	返回容器 v 的最后一个元素
v.insert(pos)	在容器 v 的 pos 位置插入一个元素
v.erase(pos)	删除容器 v 中 pos 位置上的元素
v.clear()	删除容器 v 中的所有元素
v.at(pos)	返回容器 v 中位置为 pos 的元素
v1 = v2	把容器 v1 的元素替换为容器 v2 中元素的副本，也就是将 v2 赋值给 v1
v1 == v2	判断容器 v1 和容器 v2 是否相等，如果 v1 与 v2 相等，则返回 true

在表 9-1 所述的这些操作函数中，最常使用的应该是 push_back() 函数了，我们通常使用它来将数据保存到 vector 容器中。当使用 push_back() 函数向 vector 容器中保存数据时，它首先接受一个跟 vector 容器数据类型相同的数据，然后将其复制为一个新元素并添加到 vector 容器的末尾，也就是“插入 (push)”vector 容器的“后面(back)”。例如：

```
// 创建一个空的保存整型数的 vector 容器
vector<int> vecSalary;
int nSalary;
// 循环读取用户输入
while(cin>>nSalary)
{
    // 将用户输入的数据保存到 vector 容器对象 vecSalary 中
    vecSalary.push_back( nSalary );
}
// 输出当前容器中的元素个数
cout<<"当前容器中的元素个数是："<<vecSalary.size()<<endl;
```

在这段程序中，首先定义了一个空的能够保存整型数的容器 vecSalary；然后通过循环从标准输入读取用户输入的数据，每循环一次就添加一个新元素到 vecSalary 容器的末尾位置，当循环结束时，vecSalary 容器就包含了所有读入的元素。

9.2.3 访问 vector 容器中的数据

既然数据已经保存到了容器中，那么接下来当然是希望能够访问容器中的数据，对其进行相应的操作。本质上，vector 容器就是一个数组，跟访问数组中的元素相似，我们可以使用“[]”运算符，以元素在容器中的次序作为下标来访问 vector 容器中的数据。例如：

```
// 将 vecSalary 容器中的第一个数据元素赋值为 3000
vecSalary[0] = 3000;
```


但是，为了保证访问元素的安全性与便利性，vector 容器更多的是使用迭代器或者 at() 函数访问容器中的数据元素。例如：

```
// 定义索引值变量，用于访问容器中的数据
vector<int>::size_type nIndex = 0;
// 循环遍历容器
for( auto it = vecSalary.begin();
    it != vecSalary.end(); ++it, ++nIndex )
{
    // 通过迭代器读取容器中的数据
    cout<<"当前工资是："<<*it<<endl;
    // 通过 at() 函数修改容器中元素的值
    vecSalary.at( nIndex ) += 1000;    // 涨工资啦！每个人加 1000 元！
    cout<<"涨工资是："<<*it<<endl;
}
```

这里，我们分别通过迭代器和 at() 函数对 vector 容器中的数据进行了读/写操作。虽然两种方式都可以对 vector 容器中的元素进行访问，但是在条件允许的情况下，为了保持代码的一致性，可以优先通过迭代器对容器中的元素进行访问。

vector 容器的使用如此简单，同时又比数组更加高效和安全，所以程序员就喜欢上了数组的最佳替代者——vector 容器。

9.3 可以保存键值对的 map 容器

虽然 vector 容器很好用，但是它保存和管理的数据都是单个的个体，例如，一个整型数、一个类的对象等。如果要使保存的数据成对出现，例如要让一个员工号和一个员工对象构成一个数据对，面对这种成对出现的数据，vector 就无能为力了。不过没有关系，STL 为这种成对出现的数据提供了一个专门的容器——map。

9.3.1 创建并初始化 map 容器

map 是 STL 中的一种关联容器，它提供了一种对数据对进行保存和管理的能力。这种数据对是一对一成对出现的，其中，第一个数据可以称为数据对的键（key），且每个键只能在 map 中出现一次；第二个数据可以称为该键的值（value）。例如，一个员工号和一个员工对象共同构成一个数据对，员工号是这个数据对的键，而员工对象就是这个数据对的值。在 map 容器的内部，它是使用一棵红黑树实现的——一种非严格意义上的平衡二叉树，如图 9-4 所示。因为这棵树具有对数据自动排序的功能，所以 map 内部所有的数据都是有序的。正是这种特性，使得 map 容器在增加和删除节点时对迭代器的影响很小，除了被操作的当前操作节点外，对其他的节点都没有什么影响。因此 map 容器特别适用于保存和管理大量数据。

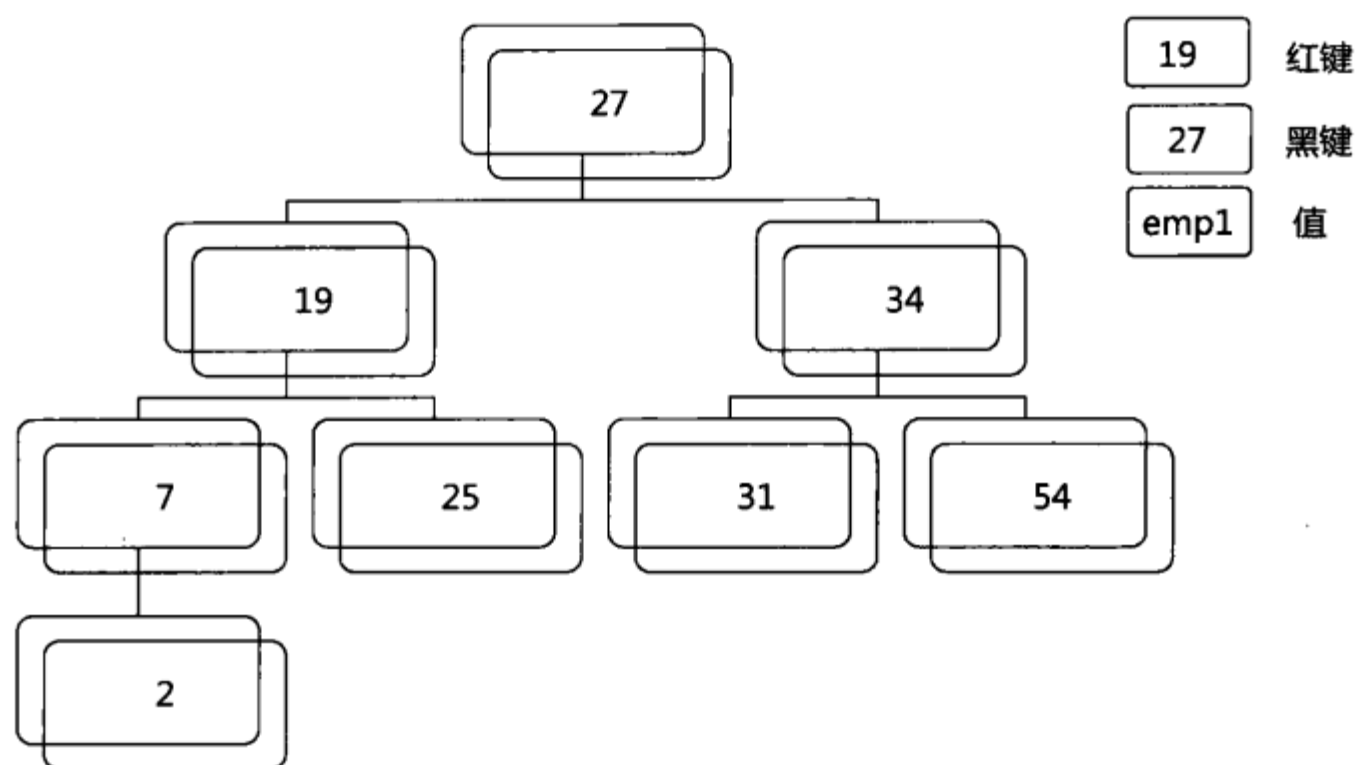


图 9-4 红黑树实现的 map 容器

跟 vector 容器的使用相似，map 容器同样是一个类模板，在使用的时候，需要给定的数据类型参数对其进行实例化以形成能够保存具体数据类型的模板类。因为管理的是数据对，所以 map 类模板需要两个数据类型作为模板参数，其中第一个是键的数据类型，而第二个就是对应的值的数据类型。例如：

```

// 引入 map 容器所在的头文件
#include <map>
// 使用 map 容器所在的名字空间
using namespace std;
// 创建一个 map 容器对象 mapEmployee
// 这个容器可以保存“int-Employee”形式的数据对
// 其中，int 是键的数据类型，Employee 是值的数据类型
map<int, Employee> mapEmployee;
  
```

经过这样简单的定义，map 容器就可以开始保存数据了。

9.3.2 将数据保存到 map 容器中

STL 中的每个容器都提供了相应的容器操作函数来对容器或者容器中的数据进行操作，map 容器当然也不例外。map 容器的常用操作函数如表 9-2 所示。

表 9-2 map 容器的常用操作函数

函 数	说 明
m.insert(pair)	先将 pair 这个数据对插入 map 容器 m 中，然后 map 容器将根据这个数据对的键对其进行排序，也就是说，map 容器始终是排序好的

函 数	说 明
m.size()	返回容器 m 中元素的个数
m.count(KEY)	判断 KEY 这个键是否在容器 m 中出现, 如果出现则返回 1; 否则返回 0
m.find(KEY)	返回容器 m 中指向键为 KEY 的数据对的迭代器
m.erase(pos)	删除容器 m 中 pos 迭代器所指向的元素
m.empty()	判断容器 m 是否为空
m.clear()	删除容器 m 中的所有元素

有了表 9-2 中的这些操作函数的帮忙, 对 map 容器的操作就简便多了。例如, 可以使用 insert() 函数向容器中插入数据对。

```
// 创建一个 Employee 对象
Employee emp1;
// 使用 pair<int, Employee> 模板类建立员工号 1 和员工对象 emp1 的联系,
// 将 pair<int, Employee> 模板类创建的对象插入 map 容器中
mapEmployee.insert(pair<int, Employee>(1, emp1));
```

这里, 我们使用 pair<int, Employee> 模板类将 1 和 emp1 组合起来, 形成一个 pair<int, Employee> 模板类的对象并插入 map 容器中, 这样员工号 1 和员工对象 emp1 就建立了一对一的映射关系。除了这种方法外, 还可以使用 map 容器的 value_type 类型来实现数据的插入。

```
// 使用 value_type 类型实现数据的插入
mapEmployee.insert(map<int, Employee>::value_type(1, emp1));
```

除了使用 insert() 函数向 map 容器插入数据之外, 更简单的还可以利用 map 容器的 “[]” 运算符, 以数据对的键/值作为索引值, 直接向 map 容器中插入数据。例如:

```
// 向 mapEmployee 容器中插入一个数据对 (1983, emp1)
mapEmployee[1983] = emp1;
```

这三种向 map 容器插入数据的方法是等效的, 可以根据自己的喜好进行选用。因为 map 容器中的所有键都要在插入后进行排序, 所以在进行插入操作的时候, 必须保证插入的键在 map 容器中是唯一的, 否则将导致插入操作失败。另外要注意的是, 因为插入数据的动作伴随数据进行排序的动作, 所以在进行插入操作的时候, 所插入的键必须是可以排序的。对于使用基本数据类型作为键的 map 容器我们无须担心, 因为它们本身已经支持 “<” 运算符。如果使用的是自定义的数据类型作为 map 容器的键, 就需要重载 “<” 运算符, 以实现 map 容器的排序。

9.3.3 根据键找到对应的值

当 map 容器中保存有数据时, 接下来的任务就是访问容器中的数据。因为 map 容器中的

数据总是成对出现的，所以对 map 容器中数据的访问就变成了按图索骥，可根据某个键找到它所对应的值进行访问。

跟 vector 容器相同的是，对于 map 容器，同样可以使用 for 循环来遍历容器中的所有数据。例如：

```
// 利用迭代器访问 map 容器中的数据
for( map<int, Employee>::iterator it = mapEmployee.begin();
    it != mapEmployee.end(); ++it )
{
    // 通过迭代器输出数据对的键和值
    cout<<"当前员工号是："<<it->first<<endl;
    cout<<"姓名："<<it->second.GetName()<<endl;
    cout<<"工资："<<it->second.GetSalary()<<endl;
}
```

map 容器的迭代器对象实际上是一个 pair 对象，它包括两个数据：it->first 和 it->second，分别代表键和跟键对应的值。这里，map 容器中保存的值是 Employee 对象，所以“it->second”实际上就是 Employee 对象，可以直接调用它的成员函数输出相应的数据。

跟 vector 容器不同的是，对于 vector 这种基于连续内存的容器，我们一般使用循环遍历的方式来访问其中的数据。对于 map 这种基于节点的容器，更多的是使用查找某个节点的方式来访问其中某个特定的数据。我们知道，map 容器中所保存的是{键，值}构成的数据对，map 容器提供了一个 find() 函数来查找某一个键，返回指向拥有这个键的数据对的迭代器，从而可以访问到这个数据对的值。例如：

```
// 定义要查找的键
int nFindKey = 1;
// 使用 find() 函数查找键，返回指向拥有这个键的数据对的迭代器
// 如果 map 容器中没有这个键，则返回指向容器末尾位置的迭代器
map<int, Employee>::iterator it= mapEmployee.find( nFindKey );

// 查看迭代器是否指向容器末尾位置，以此判断是否找到相应的数据对
if (mapEmployee.end() == it )
{
    // 如果迭代器指向容器末尾位置，就表示没有找到对应的数据对
    cout<<"无法找到键为"<<nFindKey<<"的数据对"<<endl;
}
else
{
    // 如果迭代器指向其他位置，则表示找到相应的数据对
    // 而 find() 函数返回的迭代器就是指向这个数据对的位置
    cout<<"找到键为"<<nFindKey<<"的数据对"<<endl;
    // 通过迭代器访问这个数据对的值，也就是 Employee 对象
    cout<<"姓名："<<it->second.GetName()<<endl;
    cout<<"工资："<<it->second.GetSalary()<<endl;
}
```

在这里，通过 find()函数在容器中查找具有某个键的数据对，如果能够找到，find()函数会返回指向这个数据对的迭代器，通过迭代器就可以访问这个数据对的键和值了。

除了访问某个特定键对应的数据对之外，对于 map 容器，有时还需要访问某个范围的数据对。例如，希望输出员工号从 1 到 1 000 的所有员工信息。

```
// 定义键的范围
int nFromKey = 1;
int nToKey = 1000;

// 用迭代器表示起始位置和终止位置
map<int, Employee>::iterator itfrom =
    mapEmployee.lower_bound( nFromKey );
map<int, Employee>::iterator itto =
    mapEmployee.upper_bound( nToKey );

// 判断是否找到正确的范围
if( mapEmployee.end() != itfrom &&
    mapEmployee.end() != itto )
{
    // 输出范围内的所有数据
    for( map<int, Employee>::iterator it = itfrom;
        it != itto; ++it )
    {
        cout<<"当前员工号是："<<it->first<<endl;
        cout<<"姓名："<<it->second.GetName()<<endl;
        cout<<"工资："<<it->second.GetSalary()<<endl;
    }
    // 删除范围内的所有数据
    mapEmployee.erase( itfrom, itto );
}
```

在这段代码中，分别使用了 lower_bound()函数和 upper_bound()函数来获得指向这个范围的起始位置和终止位置的迭代器，然后通过这两个迭代器所界定的范围来对某个键范围内的数据进行访问。

按图 (key) 索 (find()) 骥 (value)，map 的使用就是这么简单。

用 STL 中的通用算法处理数据

程序的两大核心任务就是管理数据和处理数据。使用 STL 中的各种容器可以对各种数据进行良好的管理，但是只管理好数据是没有用的，重要的是还要对数据进行处理，得到最终的结果，而这正是 STL 通用算法的用武之地。STL 提供了大量的通用算法，利用这些算法可以轻松完成对数据的常见处理，比如数据的查找、排序和填充等，并且算法的效率往往比自己动手实现的相同算法的效率要高。重要的是，STL 库中的这些算法都以迭代器类型为参数，这就和数据所在容器的具体实现相互分离了，我们可以把一个排序算法用于 vector 容器，也同样可以将这个排序算法用于 map 容器。这样就实现了算法的大小通吃——不管数据保存在什么容器，也不管容器中的数据到底是什么数据类型，算法都能够处理。这也是为什么 STL 中的算法称为通用算法（generic algorithm）的原因。

既然有这么简单又高效的东西，我们没有理由不用啊！

10.1 STL 算法中的“四大帮派”

STL 中的通用算法众多，按照是否改变容器中元素的顺序和是否修改数据元素，算法也各自拉帮结派，形成了 STL 算法中的“四大帮派”，如图 10-1 所示。

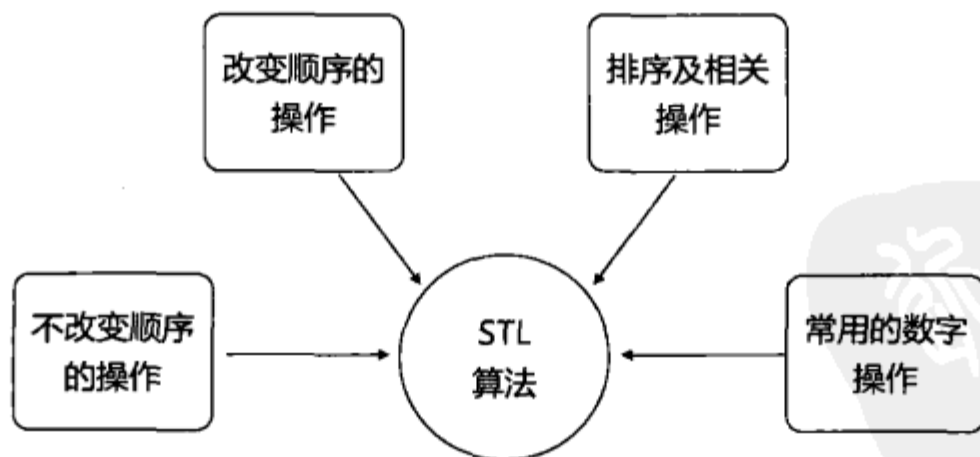


图 10-1 STL 算法的“四大帮派”

1. 不动帮：不改变顺序的操作

这一帮派的算法不会改变容器中数据元素的顺序，也就是说，它们不会改变容器中各个数据元素的位置，而只可能对数据元素的值进行访问，所以称为“不动帮”。例如，for_each()

算法就是逐个遍历容器中的数据元素并对其进行访问，读取或者修改各个数据元素的值。在 STL 中，这类算法还有查找 (find)、邻近寻找 (adjacent find)、计数 (count)、不匹配 (mismatch)、相等 (equal)、搜索 (search) 等。

2. 扰民帮：改变顺序的操作

跟“不动帮”算法只是安静地访问数据元素不同，“扰民帮”的算法专门干一些扰民的事情，它们不仅会访问数据元素的值，还会改变容器中数据元素的顺序。例如，copy() 算法可以将一个容器中的数据元素复制到另外一个容器中，从而改变目标容器中元素的顺序。这类算法通常是对数据元素进行复制变换操作，常用的算法有拷贝 (copy)、交换 (swap)、变换 (transform)、替换 (replace)、填充 (fill) 等。

3. 排序帮：排序及相关操作

因为排序算法的重要性和普遍性，所以这类算法身份比较显赫，看不起“不动帮”和“扰民帮”的算法，自己独立起来形成专门的“排序帮”。如果容器中保存的是基本数据类型的数据，比如 int 或 string 等，而这些基本数据类型已经重载了“<”运算符，则可以直接使用 sort() 算法排序容器中的数据。如果容器中保存的是通过 class 或 struct 关键字定义的自定义数据类型，则通过重载这些自定义数据类型的“<”运算符同样可以利用 sort() 算法实现自定义数据类型的排序。更进一步地，还可以通过给 sort() 算法提供表示排序规则的函数或者函数对象，对排序算法进行自定义以实现比较特殊的排序规则。除了排序算法，STL 还提供了很多跟排序相关的算法，例如：二分搜索 (binary search)、合并 (merge)、堆操作 (heap operations)、最大最小 (minimum and maximum)、词典比较 (lexicographical comparison) 等。

4. 数字帮：常用的数字操作

这一帮派是 STL 算法中的小帮派，它们都是跟数字打交道的算法。例如，聚集 (accumulate)、内部乘积 (inner product)、局部和 (partial sum)、邻近不同 (adjacent difference) 等。在开发实践中，这类算法较少用到。

10.2 容器元素的查找与遍历

面对容器中保存的大量数据，很多时候我们只是走过来瞧一瞧、看一看，从中挑选几个感兴趣的数据，或者没有那么多工夫进行挑选，一股脑儿地将容器中的数据都遍历一遍进行处理。为了完成对这些数据的处理，需要对容器元素进行查找和遍历。

10.2.1 用 for_each() 算法遍历容器中的数据元素

容器是用来保存大量数据的。很多时候我们希望能够遍历容器中的每个数据，对每个数

据都进行某种处理。

对每个数据都进行处理？可以使用 for 循环啊，还用什么算法啊？

没错，使用 for 循环，可以循环遍历容器中的每个数据元素，但是这种方式在 STL 看来仍显“粗鲁”：使用 for 循环，需要定义一个额外的循环控制变量，让它指向容器中的各个元素；需要小心地控制这个循环控制变量，防止访问越界；对容器中数据元素的访问，需要通过循环变量间接地进行。这些在 STL 看来都妨碍了访问数据的“优雅”，使容器中数据的访问变得笨拙而不自然。于是，STL 提供了一个 for_each() 算法可以“优雅”地遍历容器中的数据元素。

for_each() 可以将一个处理方法应用到一个容器的某个范围内的每个元素，以此来代替 for 循环遍历容器中的每个元素并对其进行处理。for_each() 算法的原型如下：

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function f);
```

STL 中的算法实质上都是一些函数模板，for_each() 算法也不例外。这里可以看到 for_each() 算法可以接受三个参数，前两个通常是需要处理的容器的迭代器，分别表示需要处理的数据的起始位置和终止位置；第三个参数就代表了对这个范围内的数据的处理方法，它可以是一个函数，也可以是一个函数对象，甚至是一个 Lambda 表达式。总之，在这个函数中要完成对范围内的每个数据元素的处理，例如，对公司中所有工资低于 2 000 元的低工资员工增加 30% 的薪水的代码如下。

```
// 利用函数定义我们对数据的处理方法，
// 低于 2000 元的工资，就增加 30%
void addsalary( int& nSalary )
{
    // 判断数据是否满足条件
    if( nSalary < 2000 )
        nSalary *= 1.3;    // 对数据进行处理
}

// 构造容器，保存数据
vector<int> vecSalary;
vecSalary.push_back(3200);
vecSalary.push_back(1983);
vecSalary.push_back(703);

// 使用 for_each() 算法对容器中的数据进行处理
for_each( vecSalary.begin(),
          vecSalary.end(), addsalary );
```

在这段代码中，使用 for_each() 算法完成了对容器中所有数据的处理，因为它指定的处理范围是容器的开始位置和末尾位置，当然也可以根据需要改变这个范围，只对某个特定范围内的数据进行处理。在执行过程中，for_each() 算法会逐个遍历整个范围内的所有数据

元素，并以这些元素为参数调用它的处理函数，也就是用这个函数处理这些元素。在这里，我们先将数据元素的处理方法定义在 `addsalary()` 函数中，通过 `addsalary()` 函数的形式参数，`for_each()` 算法将容器中的每个数据逐个传入 `addsalary()` 函数，然后在 `addsalary()` 函数中对数据进行处理。比如，`addsalary()` 函数会判断工资是否低于 2 000 元，如果低于 2 000 元就增加 30% 的薪水。实际上，这都是对容器中每个数据元素的操作。`for_each()` 算法对数据的处理如图 10-2 所示。

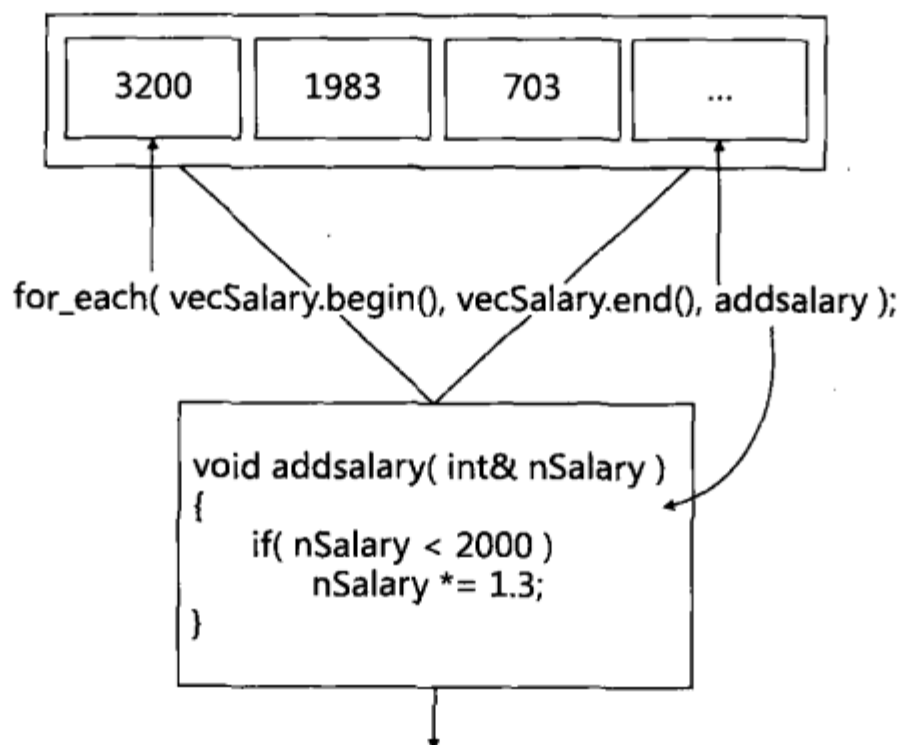


图 10-2 `for_each()` 算法对数据的处理

更形象地讲，`for_each()` 算法更像一个做化学实验时的漏斗装置，可以通过指定容器的范围来决定往这个漏斗中倒入什么数据。而 `for_each()` 算法中的处理函数就是具体的漏斗，它决定什么东西可以留下，什么东西可以漏掉，以此来完成对倒入漏斗中的数据的具体处理。更进一步讲，还可以随时更换 `for_each()` 算法中的漏斗，对数据进行不同的处理，这也体现了算法的通用性。

10.2.2 用 `find()` 和 `find_if()` 算法实现线性查找

很多时候，我们并不需要处理容器中的所有数据，更多的是希望从容器的大量数据中找到我们感兴趣的数据并进行处理。在那么多数据中，能找到我们需要的数据并不容易，而有了 STL 帮忙，就不需要众里寻她千百度，只需要 `find()` 算法就可。

STL 中的 `find()` 算法可以从一个容器的某个范围中查找具有某个特定数值的数据，它的函数原型如下：

```
template <class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last,
    const T& value );
```

从这里可以看到，find()算法可以接受三个参数。其中前两个参数都是某个容器的迭代器，用于指定查找的起始位置和终止位置；第三个参数就是要查找的内容，它的数据类型跟容器中的数据类型相同。如果 find()算法在容器中找到了相同的数据，那么返回指向这个数据元素的迭代器；如果没有找到，那么返回的迭代器就指向这个容器的末尾位置。例如，可以从一个保存商品信息的 vector 容器中查找某个商品，以此来判断这个商品是否存在。

```
// 创建保存商品信息的 vector 容器
vector<string> vecGoods;
// 向容器中添加商品
vecGoods.push_back("Eraser");
vecGoods.push_back("Book");
vecGoods.push_back("Pen");
// 定义要购买的商品
string strGood = "Pencil";
// “老板，你们这有没有卖铅笔的啊？”
// “稍等，让我找找看！”
vector<string>::iterator it = find( vecGoods.begin(),
    vecGoods.end(), strGood );
// 如果 find()函数返回的迭代器没有指向容器的末尾，
// 那就是找到想要的东西了
if( it != vecGoods.end() )
{
    cout<<"恭喜，本店提供"<<strGood<<endl;
}
else
{
    cout<<"抱歉，本店不提供"<<strGood<<endl;
}
```

这里，利用 find()算法在容器的整个范围内从 vecGoods.begin()到 vecGoods.end()查找“Pencil”数据，然后根据算法的不同返回值就可以判断这个数据是否在这个容器中，如果在容器中，还可以通过迭代器获得它的具体位置。

在容器中查找数据时，我们并不一定想从容器中找到某个特定的数据，更多时候，我们希望能够找到符合某种条件的数据。例如，从一个保存学生成绩的 vector 容器中，希望找到及格的分数，也就是大于 60 的数据。面对这种情况，find()算法就无能为力了，好在 STL 提供了它的升级版——find_if()算法。利用 find_if()算法可以对查找的规则进行自定义，从容器找出符合某种条件的数据，也就是我们真正感兴趣的数据。find_if()算法的原型如下：

```
template <class InputIterator, class Predicate>
InputIterator find_if ( InputIterator first, InputIterator last,
    Predicate pred );
```

跟 find()算法相似，find_if()算法同样用前两个参数来指定查找的范围；它们的区别是，

find_if()算法的第三个参数用来指定查找规则，它可以是一个返回值为 bool 类型的函数，也可以是一个函数对象。在这个函数中，可以对查找规则进行自定义，也就是定义希望找到什么样的数据。例如：

```
// 使用函数定义查找规则
// 如果分数大于等于 60，就返回 true，否则返回 false
bool ispass( int n )
{
    return n >= 60;
}

// 定义保存成绩的容器
vector<int> vecScores;

// 将分数保存到容器中
vecScores.push_back(72);
vecScores.push_back(54);
vecScores.push_back(87);

// 定义查找的起始位置
vector<int>::iterator it = vecScores.begin();
// 利用循环，逐个查找容器中符合条件的数据
do
{
    // 在容器中查找符合条件的数据元素，
    // 其中 ispass 定义了查找的条件
    it = find_if(it, vecScores.end(), ispass );
    if ( vecScores.end() != it )
    {
        // 输出查找到的符合条件的数据
        cout<<"找到及格分数："<<(*it)<<endl;
        // 将迭代器指向下一个位置，从新的位置开始下一次查找
        ++it;
    }
    else
    {
        // 如果没有找到，就退出循环
        break;
    }
} while( true );
```

在这段代码中，通过 find_if() 来查找符合某个条件的数据元素，而不是查找某个特定的元素。通过 ispass() 函数来定义查找的条件，find_if() 算法会遍历容器中的每个数据元素，并将其作为参数调用 ispass() 函数。如果这个条件函数返回 true，则表示这个元素符合条件，是我们需要查找的数据，find_if() 算法就会返回指向这个数据元素的迭代器；反之，find_if() 算法就会认为这个数据不符合查找条件，不是我们想要找的数据，它就会跳过这个数据，开始检查容器中的下一个数据。如此循环，直到 find_if() 算法检查完容器中的所有数据，返回指向容器末尾的迭代器。

除了 find() 算法和 find_if() 算法之外，STL 中的查找算法还有多个变种。比如，find_end() 算法能够从容器的末尾位置开始查找符合条件的数据；find_first_of() 算法会查找容器中某个元素首次出现的位置；而 adjacent_find() 算法会查找容器中重复出现的数据。跟查找算法在容器中查找单个符合条件的数据相近，STL 还提供了搜索算法，也就是 search() 算法和 search_n() 算法，它们可以用于在容器中搜索相匹配的子串数据。STL 中跟查找相关的算法有很多，它们各有各的适用场景，我们可以根据需要进行选择合适的查找算法。

使用查找算法，不仅是为了找到某个符合条件的数据，更多时候还需要对这些数据进行处理，比如删除某些符合条件的数据，或者用其他的数据替换这些符合条件的数据。STL 提供了相应的算法来帮助我们快速完成这些常见的数据处理任务。比如，STL 中的 remove() 算法可以删除 (remove) 容器中所有符合某个条件的数据，而 replace() 算法则可以使用其他的数据替换 (replace) 容器中符合某个条件的数据。利用这些算法，我们对数据的日常处理就简单多了。

下面还是来看一个实际的例子。考试结束后，所有考试成绩都保存在 vecScore 容器中。在对考试成绩的处理中，为了让考试成绩好看一点，还需要对考试成绩修饰打扮一番。首先需要删除其中的缺考成绩（如果学生缺考，则成绩记为-1），然后将所有不及格的成绩替换为及格成绩。使用 STL 中的 remove() 算法和 replace() 算法，就可以轻松完成这样一个比较复杂的数据处理任务。

```
// 判断分数是否不及格的函数
bool isfail(int nScore)
{
    return nScore < 60 ? true : false;
}

// 保存学生成绩的容器
vector<int> vecScore;
// 将学生成绩保存到容器中
vecScore.push_back(82); // 及格成绩
vecScore.push_back(-1); // 缺考成绩
vecScore.push_back(32); // 不及格成绩
// ...

// 删除缺考成绩
// 也就是删除 vecScore 容器中所有值为-1 的数据
// 因为删除元素后，容器的结束位置发生了变化，
// 所以使用一个新的迭代器来保存容器的结束位置
vector<int>::iterator itend =
    remove( vecScore.begin(), vecScore.end(), // 删除的范围
            -1);                               // 需要删除的值

// 替换不及格成绩
```

```

// 也就是使用 60 这个新的数据值替换容器中所有符合 isfail 条件的数据
replace_if(vecScore.begin(), itend,    // 替换的范围
           isfail,                      // 替换的条件
           60);                        // 替换后的新数据值

// 输出处理后的成绩，这样子看起来漂亮多了
// 在这里，我们使用 auto 关键字作为循环索引值 it 的数据类型，
// 编译器会根据它的首次赋值推定它的真实数据类型，
// 这里，it 首次被赋值为 vecScore.begin()，所以它会被推定为 vector<int>::iterator
cout<<"处理后的成绩是："<<endl;
for(auto it = vecScore.begin();
    it != itend; ++it)
    cout<<*it<<endl;

```

10.3 容器元素的复制与变换

除了对容器中已有的数据进行处理之外，很多时候还需要施展乾坤大挪移，将一个容器中的数据经过复制或者变换存放到另外一个容器中，这就是容器元素的复制与变换。

10.3.1 复制容器元素：copy()算法

有时候，我们需要将一个容器中的元素复制到另外一个容器中去，来完成数据的备份或者进行其他处理。要完成容器元素的复制，首先要通过 for 循环将数据元素从一个容器中取出来再添加到另外一个容器中去，或者使用赋值操作符(=)来完成容器的复制。这两种方法都可行，但是都有一定的局限性：使用 for 循环太过烦琐；而使用赋值操作符只能整个复制容器，缺乏灵活性。这些缺点对于 STL 追求“简约而不简单”的优雅人士来说是绝不允许的。于是 STL 提供了通用的 copy()算法来完成容器元素的复制。在 STL 中，copy()算法的原型如下：

```

template <class InputIterator, class OutputIterator>
    OutputIterator copy ( InputIterator first, InputIterator last,
                          OutputIterator result );

```

copy()算法可以接受三个参数，前两个参数表示需要复制的源容器的起始位置和终止位置，它们共同定义了需要复制的数据元素的范围。第三个参数则是目标容器的起始位置。换句话说，也就是将源容器中某个范围内的数据移到目标容器的起始位置。例如，在学校中，老师常常要将多个班级的学生成绩统计到一起，形成一张学生成绩总表。以前，老师是辛苦地将一个个成绩抄写到新的成绩表上，现在使用 copy()算法，事情就简单了。

```

// 保存 C1 和 C2 班级成绩的容器
vector<int> vecScoreC1;
vector<int> vecScoreC2;

// 对容器进行操作，将成绩保存到各自的容器中

```

```
// ...

// 保存所有成绩的总容器
vector<int> vecScore;
// 根据各个分容器的大小，重新设定总容器的容量，
// 使它可以容纳即将复制进来的数据
vecScore.resize( vecScoreC1.size() + vecScoreC2.size() );
// 将第一个容器 vecScoreC1 中的数据复制到 vecScore 中
vector<int>::iterator lastit = copy(vecScoreC1.begin(), vecScoreC1.end(),
vecScore.begin() );
// 将第二个容器 vecScoreC2 中的数据追加到 vecScore 的末尾
copy(vecScoreC2.begin(), vecScoreC2.end(), lastit );
```

在以上这段代码中，首先，分别使用了两个容器来保存两个班级的成绩，定义了另外一个容器 `vecScore` 来保存两个班级总的成绩。为了让这个总容器能够容纳所有的成绩，我们使用 `resize()` 函数调整了 `vecScore` 容器的容量。接着，利用 `copy()` 函数将 `vecScoreC1` 中的所有数据从开始位置到末尾位置，复制到 `vecScore` 容器的开始位置。`copy()` 函数复制完成后，它会返回一个目标容器的迭代器，它指向的是所有复制进来的数据的末尾位置，而这个迭代器正是我们第二次复制的目标位置。利用两次复制，可以成功地将两个容器中的数据复制到新容器中，并实现数据的连接，如图 10-3 所示。

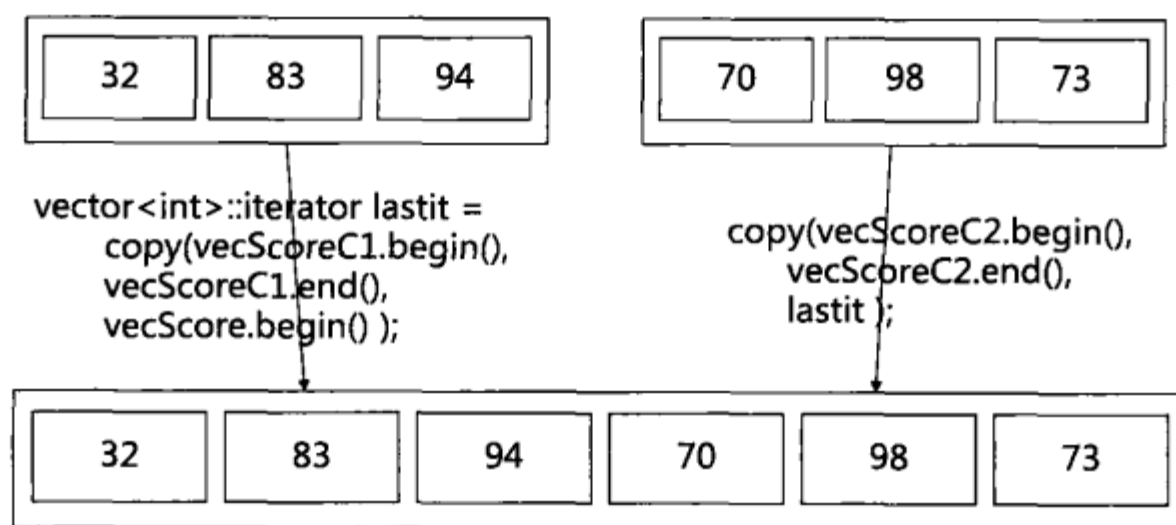


图 10-3 `copy()` 算法实现的数据连接

`copy()` 函数可以将某个容器中的数据正向复制到另外一个容器中，也就是说，这些复制的元素是从复制的目标位置开始逐个向后放置到目标容器中的。但是有的时候，我们需要将复制的元素从后向前放置到目标容器中，这时该怎么办呢？别着急，STL 想到了这一点，专门提供了能够逆向复制元素的 `copy_backward()` 算法，也称为逆向复制，它的原型如下：

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward ( BidirectionalIterator1 first,
                                       BidirectionalIterator1 last,
                                       BidirectionalIterator2 result );
```

`copy_backward()` 算法的使用跟 `copy()` 算法的使用非常相似，它同样可以接受三个参数，

前两个参数表示需要复制的数据元素的范围；而跟 `copy()` 算法不同的是，它的第三个参数是指向目标容器某个位置的迭代器，这个位置将是这些元素复制完成后的结束位置，换句话说，这些被复制进来的元素将从这个位置开始逐个向前放置到目标容器中。这里需要注意的是，如果是同一个容器内的复制，那么这个位置不能指向其源范围内的某个位置。例如，高校的大学生已经很多了，可是各个高校还在不停地扩招，今年扩招一倍，明年再扩招一倍，这就是 `copy_backward()` 带来的恶果：

```
// 保存学生对象的容器
vector<Student> vecStudent;
// 将数据保存到容器中...
// 扩大容器的容量为原来的两倍
// 这样容器中前半部分是已有的数据，后半部分是默认生成的数据
vecStudent.resize( vecStudent.size() * 2 );
// 将前半部分已有的数据复制到后半部分，替换掉后半部分默认的数据
copy_backward(vecStudent.begin(),
              vecStudent.begin() + vecStudent.size() / 2,
              vecStudent.end() );
```

在这段程序中，首先定义了一个用于保存学生对象的容器，因为要扩招一倍，所以利用 `resize()` 函数将容器扩容为原来的两倍。这时，容器中前半部分是已有的数据，而后半部分还是默认生成的数据。其次，利用 `copy_backward()` 算法将前半部分的数据复制到自身容器的后半部分。这样，利用 `copy_backward()` 算法可以轻松地将容器及其中的数据扩展一倍。扩招这么容易，难怪高校要不停地扩招了。

10.3.2 合并容器元素：merge() 算法

除了使用 `copy()` 算法烦琐地逐个复制元素实现两个容器的合并之外，STL 还提供了一个 `merge()` 算法完成这个功能。顾名思义，`merge()` 算法就是专门用来将两个源容器中的数据合并（merge）到目标容器的算法。在 STL 中，`merge()` 算法的原型如下：

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result );
```

其中，`(first1,last1)` 和 `(first2,last2)` 分别表示两个源容器数据范围的迭代器，而 `result` 则表示合并到目标容器的起始位置。使用 `merge()` 算法可以轻松地将两个排序后的容器中的所有数据合并到一个新的目标容器中。例如，可以利用下面的代码将 `vecScoreC1` 和 `vecScoreC2` 两个容器中的数据经过排序后合并到目标容器 `vecScore` 中。

```
// 在使用 merge() 算法进行合并之前，必须先使用 sort() 算法对两个源容器中的数据进行排序
// 关于 sort() 算法的使用，接下来的 10.4.1 小节将有详细的介绍
sort(vecScoreC1.begin(), vecScoreC1.end());
sort(vecScoreC2.begin(), vecScoreC2.end());
```



```

// 调整目标容器的大小，以便容纳合并进来的数据
vecScore.resize(vecScoreC1.size() + vecScoreC2.size());
// 使用 merge() 算法将 vecScoreC1 和 vecScoreC2 两个容器中的数据
// 合并到目标容器 vecScore 中
merge(vecScoreC1.begin(), vecScoreC1.end(),      // 第一个容器的范围
      vecScoreC2.begin(), vecScoreC2.end(),      // 第二个容器的范围
      vecScore.begin());                        // 目标容器的起始位置

```

当两个容器的数据进行合并时，两个容器中往往有一些相同的数据，这些相同的数据在合并之后往往是一些冗余元素。所以很多时候，我们在进行数据合并的同时，还希望保留一份两个容器中相同的数据，从而保持目标容器中数据的一致性。为了完成这项任务，STL 提供了单独的 `set_union()` 算法。`set_union` 算法在合并两个源容器中的数据时，如果遇到两个源容器都有的数据，则只合并一份数据到目标容器中，这样目标容器中就不会有相同的冗余元素了。例如，一个文具商店有两份商品清单，分别记录文具类商品的 `vecStationaries` 和记录办公用品类的 `vecOfficeSupplies`。因为这两类商品比较相近，所以在这两份清单中有些商品是相同的，比如，“Pen” 这个商品既在文具类商品清单 `vecStationaries` 中，也在办公用品类商品清单 `vecOfficeSupplies` 中。现在老板想将这两份商品清单合并到总的商品清单 `vecGoods` 中，而又不想“Pen” 在总商品清单 `vecGoods` 中出现两次，`merge()` 算法没法完成这任务，只有请 `set_union()` 算法出马了。

```

// 定义总商品清单
vector<string> vecGoods;

// 文具类货物清单
vector<string> vecStationaries;
vecStationaries.push_back("Pen");
vecStationaries.push_back("Erase");

// 办公用品类商品清单
vector<string> vecOfficeSupplies;
vecOfficeSupplies.push_back("Folder");
vecOfficeSupplies.push_back("Pen");

// 根据源容器的数据多少调整目标容器的大小
vecGoods.resize( vecStationaries.size() + vecOfficeSupplies.size() );

// 对源容器进行排序
sort(vecStationaries.begin(), vecStationaries.end());
sort(vecOfficeSupplies.begin(), vecOfficeSupplies.end());

// 使用 set_union() 算法将源容器中的数据合并到目标容器中
// set_union() 算法返回的是指向合并后的目标容器中最后一个数据的迭代器
vector<string>::iterator itend =
    set_union( vecStationaries.begin(), vecStationaries.end(),
              // 第一个容器的范围
              vecOfficeSupplies.begin(), vecOfficeSupplies.end(), // 第二个容器的范围

```

```

        vecGoods.begin()); // 目标容器的起始位置
// 输出合并后的商品
for(auto it = vecGoods.begin(); it != itend; ++it)
    cout<<*it<<endl;

```

实际上，`set_union()`算法是用于计算两个容器的并集。相似地，STL 还提供了 `set_difference()`算法用于计算两个容器的差集，而 `set_intersection()`算法则可以用于计算两个容器的交集。有了这些关于两个容器的集合运算算法，当处理来自两个容器的数据时就可以更加简单。

10.3.3 变换容器元素：transform 函数

在容器之间移动数据元素的时候，不仅希望能够进行元素的简单复制，有时候还希望能够在复制元素的同时对元素进行某些操作。例如，希望将某个容器中的数据作加法操作后移动到另外一个容器中，甚至希望同时操作两个容器的数据后将操作的结果保存到目标容器中。STL 中的 `copy()`算法虽然能够移动数据，但是无法在移动过程中对数据进行操作。要想在移动数据的时候进行操作，只有使用 STL 中的乾坤大挪移——`transform()`算法。`transform()`算法的原型如下：

```

template < class InputIterator, class OutputIterator, class UnaryOperator >
OutputIterator transform ( InputIterator first1, InputIterator last1,
                          OutputIterator result, UnaryOperator op );

template < class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperator >
OutputIterator transform ( InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, OutputIterator result,
                          BinaryOperator binary_op );

```

`transform()`算法有两个版本，其中，第一个版本的 `transform()`算法可以接受 4 个参数，前两个参数用于指定输入容器的范围；第三个参数用于指定目标容器的起始位置；最后一个参数就是要在移动过程中执行的动作，也就是算法的操作函数，它可以是一个函数或者函数对象。第二个版本的 `transform()`算法可以同时两个容器作为输入，所以它在第一个版本的基础上增加了一个参数 `first2`，用来表示第二个输入容器的起始位置，其他参数的意义跟第一个版本的参数意义相同。`transform()`算法在执行过程中，它首先会逐个遍历输入容器范围内的数据元素，并将它作为参数调用算法中指定的操作函数，然后将操作函数的返回值作为处理的结果保存到算法的目标容器中。`transform()`算法就这样在移动数据元素到目标容器的过程中完成了对数据元素的处理，最终保存到目标容器中的数据并不是原始的数据，而是经过处理后的结果数据。

利用 `transform()`算法，可以在移动数据的时候执行定义的操作，完成从一种数据到另一种数据的华丽变身。例如，老师在统计学生成绩的时候，由于这个班的成绩实在太差，不得

不将快要及格的分数调整为及格分数，以免学生回去挨家长的板子，于是老师用 `transform()` 算法进行了一次乾坤大挪移，实现了一次从差成绩到好成绩的华丽变身。

```
// 定义移动数据过程中的操作函数
// 操作函数的参数类型就是输入容器中的数据类型
int increase( int nScore )
{
    // 对数据进行处理
    // 将 30 到 60 之间的数据调整为 60，
    // 也就是让快要及格的同学都拿及格分数
    if( nScore > 30 && nScore < 60 )
        nScore = 60;

    // 返回处理后的结果数据
    return nScore;
}

// 定义保存学生成绩的容器
vector<int> vecScoreMath;
// 将成绩保存到容器中
vecScoreMath.push_back(26);
vecScoreMath.push_back(42);
vecScoreMath.push_back(72);

// 对容器中的数据进行处理，将不及格的分数调整为及格分数
transform(vecScoreMath.begin(), vecScoreMath.end(),           // 输入数据的范围
          vecScoreMath.begin(),                                // 保存处理结果的容器的起始位置
          increase );                                           // 对数据进行处理的操作函数
```

在这段代码中，我们将保存学生成绩的容器作为 `transform()` 算法的输入，同样也将这个容器作为目标容器来保存最终的处理结果。当然，也可以选择其他容器作为目标容器来实现数据元素的挪移。在 `transform()` 算法中，我们指定了处理函数为 `increase()` 函数，这样在对每个数据进行处理时，算法就会以这个数据为参数来调用 `increase()` 函数。在 `increase()` 函数中，我们对数据做了具体的处理并将处理结果作为返回值返回。`transform()` 算法会将 `increase()` 函数的返回值作为处理结果保存到目标容器中，这样就完成了数据的移动与处理。

`transform()` 除了可以操作容器中的数据元素之外，还可以接受第二个容器作为输入，将两个容器中的数据元素联合起来进行操作。例如，学生的数学成绩和英语成绩分别保存在两个容器中，现在班主任要将两个成绩总和起来，看看学生的综合成绩。

```
// 定义移动数据过程中的操作函数
// 这里，将两个分数相加起来作为函数返回值返回
int add( int nScoreMath, int nScoreEng )
{
    return nScoreMath + nScoreEng;
}
```

```

// 保存学生分科成绩的容器
vector<int> vecScoreMath;
vector<int> vecScoreEng;
// 将分数保存到容器中

// 定义保存综合成绩的容器
vector<int> vecScore;
// 改变容器的容量，让它有足够的空间保存结果
vecScore.resize( vecScoreMath.size() );
// 将 vecScoreMath 和 vecScoreEng 容器中的成绩相加，
// 保存到结果容器 vecScore 中
transform(vecScoreMath.begin(), vecScoreMath.end(), // 输入数据的范围
          vecScoreEng.begin(), // 第二个输入数据的起始位置
          vecScore.begin(), // 保存处理结果的容器的起始位置
          add ); // 对数据进行处理的操作函数

```

跟上面的例子相似，这里在调用 `transform()` 算法的时候，只是添加了第二个容器作为输入容器。`transform()` 算法在执行的过程中会同时访问这两个容器中的数据，并将它们作为参数调用 `add()` 函数；`add()` 函数对分别来自 `vecScoreMath` 和 `vecScoreEng` 这两个容器的数据进行处理后，只是简单地计算两个数的和，并将处理结果作为函数返回值返回，而 `transform()` 算法就会将这个返回值作为最终的处理结果保存到目标容器中。这样，就将两个容器中的数据相加到了目标容器中。

当然，现实的情况并不这么简单，`transform()` 算法给我们提供了一个在挪动数据的过程中对数据进行处理的机会，至于到底如何处理数据才能达到要求，就全靠自己的才智了。

10.4 容器元素的排序

这是一个有秩序的世界：买票需要按照先来后到排序；学生需要按照成绩好坏排序；上体育课需要按照身高高矮排序；连走路都需要按照官位大小排序。

既然 C++ 世界是现实世界的反映，那么自然也少不了排序，各种排序的方法各式各样、五花八门，有什么冒泡排序、选择排序、插入排序、希尔排序、快速排序等，结果可能连个一二三都还没有排出来，人都已经晕了。

好在 STL 提供了 `sort()` 排序算法，成为了我们的大救星。

10.4.1 使用 `sort()` 算法对容器中的数据进行排序

跟 STL 中的其他算法一样，`sort()` 同样是一个函数模板，它的函数原型如下：

```

template <class RandomAccessIterator>
void sort ( RandomAccessIterator first, RandomAccessIterator last );

```

sort()算法可以接受两个参数,分别用于指定需要进行排序的数据的起始位置和终止位置。只要确定了排序的范围,sort()算法就会将数据元素按照从小到大的顺序排序,至于具体的排序和数据的移动,就无须我们操心了,sort()算法会处理好一切。例如,最常见的按照姓名的排序的代码如下。

```
// 创建保存姓名的容器
vector<string> vecName;
// 将姓名保存到容器中
vecName.push_back("Jiawei");
vecName.push_back("ChenLiangqiao");
vecName.push_back("Jiajunpeng");
vecName.push_back("Xibei");
// 使用 sort() 算法对容器中的数据进行排序
// 这里需要指定排序的起始位置和终止位置
sort( vecName.begin(), vecName.end() );
```

sort()算法执行完成后,就会发现容器中的数据已经是按照从小到大的顺序排好的。一个简单的函数调用就可完成所有数据的排序工作,从而将这些苦命的程序员从那些让人头疼的冒泡排序、插入排序之类的复杂算法中解放出来。因此,我们从内心里发出赞叹:STL,你真是我们的大救星。

默认情况下,sort()算法处理后的数据是按照从小到大的顺序排列的,如果想按照从大到小的顺序排列数据,则可以在容器排序完成之后,调用 reverse()算法将容器翻转即可。例如:

```
// 将排序完成的容器进行翻转,以实现从大到小排序
reverse(vecName.begin(), vecName.end() );
```

对于基本数据类型,因为这些数据类型都已经重载了小于运算符(<),具备了小于运算的能力,所以保存基本数据类型的容器都可以使用 sort()算法进行排序,无须进行任何额外的工作。但是,如果容器中保存的是自定义的数据类型,比如某个类的对象、某个结构体等,就需要重载这个类型的小于运算符(<),这样,sort()算法才可以使用这些数据的小于运算符对容器中的数据进行排序。例如,容器中保存的是自定义的表示矩形的 Rect 对象,我们希望按照矩形的面积进行排序,其代码如下:

```
// 创建表示矩形的 Rect 类
class Rect
{
public:
    // 构造函数,设定矩形的长和宽
    Rect( float fW, float fH )
        : m_fW(fW), m_fH(fH) {};
    // 重载操作符 "<", 用于比较两个矩形的大小
    // 它的参数是另外一个 Rect 对象的引用
    bool operator < (const Rect& rRectOther )
    {
        // 计算两个矩形的面积
```

```

        float fArea = m_fW * m_fH;
        float fAreaOhter = rRectOther.m_fW * rRectOther.m_fH;
        // 返回两个面积比较的结果作为两个矩形比较的结果
        return fArea < fAreaOhter;
    };
    // 矩形的属性：长和宽
public:
    float m_fW;
    float m_fH;
};

// 创建保存 Rect 对象的容器
vector<Rect> vecRect;
// 将 Rect 对象添加到容器中
vecRect.push_back( Rect(3, 4) );
vecRect.push_back( Rect(6, 7) );
vecRect.push_back( Rect(8, 1) );
// 对容器中的数据进行排序
sort( vecRect.begin(), vecRect.end() );

```

在这段程序中，我们在容器中保存的是自定义的 Rect 类的对象，为了让这种数据类型也能够支持 sort() 算法，重载了 Rect 类的小于运算符 (<)。在这个重载的运算符中，定义了两个 Rect 对象比较的规则：首先计算了两个矩形的面积，然后将两个面积比较的结果作为两个矩形比较的结果，也就是说，哪个矩形的面积大，就认为这个矩形大。sort() 算法在对容器中的 Rect 对象进行排序的时候，就会调用这个运算符对两个 Rect 对象进行比较，以此来决定它们之间的大小。所以，当利用 sort() 算法对自定义的数据类型进行排序时，必需重载这种类型的小于运算符 (<)，让 sort() 算法可以决定自定义数据类型之间的大小。

容器中的数据排序完成之后，还可以进行更多的处理，比如，可以使用 lower_bound() 算法和 upper_bound() 算法获得容器中小于某个值或者大于某个值的临界点位置；可以使用 equal_range() 算法获得容器中所有等于某个值的数据范围。有了这些算法的帮助，对排序完成后容器中数据的进一步处理将更加简单。还是以 10.2.2 小节中处理学生成绩为例，当所有成绩都保存到 vecScore 容器中并用 sort() 算法进行排序之后，需要统计出所有及格的学生总人数及刚好及格的学生人数：

```

// 保存学生成绩的容器
vector<int> vecScore;

// 将学生成绩保存到 vecScore 容器中
// ...
// 对容器中的数据进行排序
sort( vecScore.begin(), vecScore.end() );

// 排序完成后，就可以开始对容器中的数据做进一步的处理

// 获取容器中大于及格分数（临界值）的临界点，
// upper_bound() 算法返回的是指向这个临界点的迭代器

```



```

vector<int>::iterator itpass =
    upper_bound(vecScore.begin(), vecScore.end(), // 容器的数据范围
        59); // 临界值

// 定义可以保存两个迭代器的 pair 对象
pair<vector<int>::iterator, vector<int>::iterator> passScores;
// 获取容器中等于某个值的数据范围
// equal_range() 算法返回的是一个 pair 对象,
// 它的 first 成员保存的是这个范围的起始位置,
// 而它的 second 成员保存的是这个范围的结束位置
passScores = equal_range(vecScore.begin(), vecScore.end(), // 容器的数据范围
    60); // 相等的数值

// 输出数据处理的结果
// 输出所有及格的分数
cout<<"所有及格的分数是:"<<endl;
for( auto it = itpass; it != vecScore.end(); ++it )
    cout<<*it<<endl;

// 利用迭代器计算符合条件的数据的数目
cout<<"所有及格的学生人数:"<<int(vecScore.end() - itpass)<<endl;
cout<<"刚好及格的学生人数:"<<int(passScores.second - passScores.first)<<endl;

```

在这里，我们利用迭代器计算符合条件的数据的数目，实际上是 `equal_range()` 算法的一个副产品。实际上，STL 提供了 `count()` 算法和 `count_if()` 算法，专门用于统计容器中符合某个条件的数据的数目。例如，可以使用这两个算法轻松完成上面代码的统计功能。

```

// 统计容器中所有及格的分数个数
// count_if() 算法使用 ispass() 函数判断当前数据是否属于及格分数,
// 如果 ispass() 函数返回的是 true, 则当前数目统计在内
int nAllPass = count_if( vecScore.begin(), vecScore.end(), // 容器数据范围
    ispass); // 判断函数
cout<<"所有及格的学生人数:"<<nAllPass<<endl;
// 统计刚好及格的学生人数
// 也就是利用 count() 算法统计容器中有多少个被统计的数值
int nPass = count( vecScore.begin(), vecScore.end(), // 容器数据范围
    60); // 被统计的数值
cout<<"刚好及格的学生人数:"<<nPass<<endl;

```

10.4.2 对排序的规则进行自定义

作为程序员，STL 的使用者，关心的并不是排序算法的具体实现，因为已经有人实现了高效的排序算法，只需要应用即可。更多时候关心的是排序的规则，关心的是如何对数据进行排序以实现业务逻辑。上面介绍的 `sort()` 算法虽然能够对容器中的数据快速方便地进行排序，但是它的排序规则是默认的从大到小的顺序。如果想对排序的规则进行自定义，就需要

用到 sort() 算法的第二个版本，它的原型如下：

```
template <class RandomAccessIterator, class Compare>
void sort ( RandomAccessIterator first, RandomAccessIterator last,
            Compare comp );
```

第二个版本的 sort() 算法在第一个版本的基础上增加了一个参数 comp，而正是这个增加的参数可以对排序的规则进行自定义。这个参数可以是一个函数或者函数对象，在这个函数中可以自定义排序的比较规则，所以这个函数也称为比较函数。这个比较函数有两个输入参数，分别是需要进行比较的两个数据，也就是容器中的某两个数据。在函数中，我们利用这两个数据按照一定的比较规则进行运算，并最终得出一个 bool 类型的结果，而这个结果就是比较函数的返回值，它代表这两个数据之间的大小关系。sort() 算法会根据这个函数的返回值最终确定这个数据在容器中的位置。例如，可以利用 sort() 算法中的比较函数重新定义容器中 Rect 对象的比较规则，让它们按照长进行排序。

```
// 定义比较函数，实现自己的比较规则
// 比较函数的两个参数的数据类型就是容器中的数据类型，而返回值类型是 bool 类型
bool sortbyH( Rect rect1, Rect rect2 )
{
    // 在比较函数中，我们定义具体的比较规则，
    // 这里比较两个矩形的长
    return rect1.m_fH < rect2.m_fH;
}
// 使用自定义的比较规则代替默认的比较规则对容器中的数据进行排序
sort( vecRect.begin(), vecRect.end(), sortbyH );
```

利用 sortbyH() 函数对比较规则进行重新定义之后，当 sort() 算法对容器中的数据进行排序时，将不再采用默认的排序规则，也就是不再调用这些数据类型的小于运算符 (<)，转而调用自己定义的排序函数对数据进行比较。因为可以完全控制比较函数的实现，从而可以把自己的比较规则实现到比较函数中，以此实现 sort() 算法比较规则的自定义。

比较规则的自定义还可以带来另外一个好处，那就是可以灵活改变比较规则，实现比较规则的多样化，想怎么排序就怎么排序。比如，如果某天老板心血来潮，要让 Rect 对象按照宽进行排序，利用 sort() 算法的比较规则自定义，也可以很快实现。

```
// 定义新的比较函数，实现新的比较规则
bool sortbyW( Rect rect1, Rect rect2 )
{
    return rect1.m_fW < rect2.m_fW;
}
// 在 sort() 中使用新的比较函数，实现新的比较规则
sort( vecRect.begin(), vecRect.end(), sortbyW );
```

在这里，我们很快将新的比较规则实现在比较函数 sortbyW() 中，然后通过调用 sort() 算法中应用这个比较函数，从而改变 sort() 算法原有的比较规则，自然排序的结果也就发生了变化。

这正是同一个 `sort()` 算法、不同的比较函数就有不同的比较结果，如图 10-4 所示。

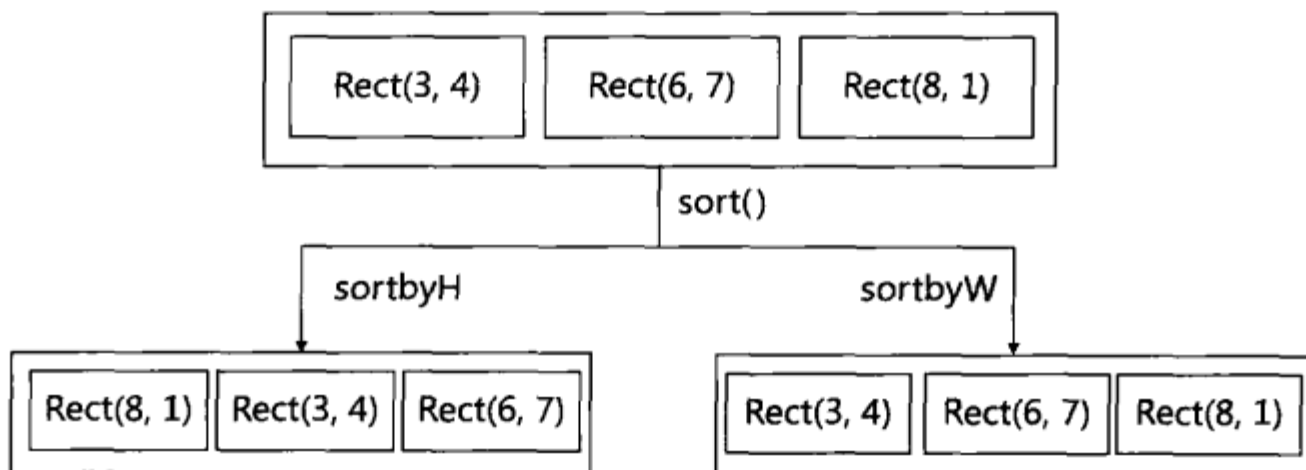


图 10-4 不同的比较函数，不同的比较结果

一般来讲，`sort()` 提供的是一个排序算法的框架，它自己维护排序算法的整个过程，例如，先从容器中取出两个数据，然后进行比较，最后根据比较的结果将数据放到容器的合适位置，在应用排序算法的时候，无须关心排序算法的复杂实现细节。但是，它将排序算法中最核心的部分——比较规则的定义留给了我们，让我们对比较规则进行自定义，以此实现同一个算法，应用不同的规则，处理不同的数据，最终达到算法的通吃天下。

获取容器中的最大值和最小值

除了对容器中的数据进行排序之外，很多时候还需要获得一个容器中所有数据的最大值和最小值。为了完成这个简单而常见的任务，STL 提供了 `max_element()` 和 `min_element()` 算法。例如，可以利用这两个算法方便地获得 `vecRect` 容器中的最大矩形和最小矩形。

```
// 利用 max_element() 算法获取 vecRect 容器中的最大值，也就是面积最大的矩形
// max_element() 算法返回的是指向这个最大值的迭代器
vector<Rect>::iterator maxit = max_element( vecRect.begin(), vecRect.end() );
// 通过指向这个最大值的迭代器访问最大值
Rect maxRect = *(maxit);
// 利用 min_element() 算法获取 vecRect 容器中的最小值
vector<Rect>::iterator minit = min_element( vecRect.begin(), vecRect.end() );
Rect minRect = *(minit);
```

跟排序算法一样，也可以通过一个函数或者函数对象作为这两个算法的第三个参数对比较规则进行重新定义，实现算法的自定义。例如：

```
// 比较两个矩形的周长，对比较规则进行重新定义
bool maxbyGirth( Rect rect1, Rect rect2 )
{
    return (rect1.m_fW + rect1.m_fH )
        < (rect2.m_fW + rect2.m_fH);
}
// 获取 vecRect 容器中周长最长的矩形
```

```
vector<Rect>::iterator maxgirthit = max_element( vecRect.begin(),
    vecRect.end(),
    maxbyGirth );
```

现在所得到的已经是周长最大的矩形了。

10.5 实战 STL 算法

STL 算法不是拿来看的，而是拿来练的。

10.5.1 “算法”老师带来的一堂别开生面的体育课

今天晴空万里，几片云彩飘在蔚蓝的天空……（此处省掉了描写 C++ 世界如何风光秀丽，STL 如何英明神武，上下形式一片大好的文字 1983 字）今天上课的是一位从 STL 中请来的新老师，他有个奇怪的名字：算法。这位“算法”老师给大家上了一堂别开生面的体育课。

这些学生实际上都是 Student 类的对象，这些学生都有自己的属性和方法。例如，都有自己的名字和身高，当然，也会报告自己的名字和身高。

```
// 学生类，描述属性和行为
class Student
{
public:
    // 默认构造函数
    Student()
    {
        m_nHeight = 0;
        m_strName = "";
    };
    // 接受姓名和身高作为参数的构造函数，
    // 可以设定对象的姓名和身高
    Student( string strName, int nHeight )
        : m_strName(strName), m_nHeight(nHeight) {};
    // 对象的行为
public:
    // 获得身高
    int GetHeight()
    {
        return m_nHeight;
    }
    // 向屏幕输出，报告名字和身高
    void ReportName()
    {
        cout<<"姓名: "<<m_strName<<"\t 身高: "<<endl;
    }
    // 对象的属性
private:
```

```

        string m_strName;    // 姓名
        int m_nHeight;      // 身高
    };

```

可怜得很，我们班只有三个同学，他们分别是：

```

// 我和我的同学们
Student st1("ChenLiangqiao", 173);
Student st2("Jiawei", 163);
Student st3("Jiajunpeng", 187);

```

上课铃响了，要开始上课了，老师把我们召集起来放到一个 vector 容器中。

```

// 老师用 push_back() 函数将我们召集到 vector 容器中
vector<Student> vecStu;
vecStu.push_back( st1 );
vecStu.push_back( st2 );
vecStu.push_back( st3 );

```

正当我们要开始上课的时候，隔壁班级的同学看到这边上课有趣，也想要过来加入。他们已经集合到另外一个容器 vecStuC2 中了，于是“算法”老师利用 copy_backward() 算法将他们合并到我们所在的 vecStu 容器中。

```

// 扩大我们所在容器的容量，为新来的同学腾出点地方
vecStu.resize( vecStu.size() + vecStuC2.size() );
// 将新同学对象复制到我们的容器中，欢迎新同学
copy_backward( vecStuC2.begin(), vecStuC2.end(), vecStu.end());

```

好了，这下人到齐了，老师一声哨响开始上课了：“全体同学都到了，立正，向右看齐，高个儿的站左边，低个儿的站右边。”这是老师让我们按照身高排队，老师还为我们提供了一个排队的规则：高个儿的站左边，低个儿的站右边。

```

// 排队的规则：高个儿的站左边，低个儿的站右边
bool sortByHeight( Student st1, Student st2 )
{
    // 两两之间比较一下身高
    return st1.GetHeight() < st2.GetHeight();
}

```

排队的规则有了，那就赶快用 sort() 算法按照这个规则排好队伍吧。

```

// 按照老师的排队规则排好队伍
sort( vecStu.begin(), vecStu.end(), sortByHeight );

```

使用 sort() 算法排队真是又快又好，只用了 0.02 秒，原来乱哄哄的队伍一下就按照身高排好了顺序。老师看我们已经排好了队，于是又开始下达命令：“开始报数。”一个 for_each() 算法让我们每个人都动了起来，各自调用自己的 ReportName() 成员函数，报出了自己的姓名和身高。

```

// 让容器中的每个对象都调用自己的 ReportName() 函数，
// 报出自己的姓名和身高

```

```
// 这里算法老师使用 mem_fun_ref() 函数将类的成员函数地址
// 构造成函数对象供我们执行
for_each( vecStu.begin(), vecStu.end(),
          mem_fun_ref(&Student::ReportName));
```

“算法”老师只用了几个简单的算法就把这群 Student 对象弄得服服帖帖，我们都很佩服他。后来又玩了捉迷藏——用 find_if() 算法找到躲起来的人——用 transform() 算法将自己改头换面扮演成其他人。这真是一堂别开生面的体育课。

10.5.2 删除容器中的冗余元素

现实世界流行减肥，各位爱美女士都想减去自己身上多余的赘肉。这股减肥风潮也很快吹到了 C++ 世界，各个容器都嚷着要删除容器当中冗余的数据元素。

还是 STL 大侠有办法，利用几个算法三下两下就把容器中的冗余元素删除了，给容器瘦了身，效果比那些所谓的“减肥茶”好多了。本来这是 STL 大侠的独家秘方，可是看各个容器求瘦心切，于是就公布出来让各个容器借助此方法也苗条起来。

很多时候，容器中都不可避免地出现冗余元素，可能是用户错误地重复输入，也可能是程序处理过程中产生的冗余元素。这就像某些人喝凉水都会长肉一样，容器中冗余元素的产生，想逃是逃不掉的。

```
// 这里的 st1 和 st3 虽然是不同的对象，但是它们的属性都相同，
// 代表了相同的意义，因而是冗余元素
Student st1("ChenLiangqiao", 173);
Student st2("Jiawei", 163);
Student st3("ChenLiangqiao", 173);
// 将这些对象添加到容器中，容器中产生了冗余元素
vecStu.push_back( st1 );
vecStu.push_back( st2 );
vecStu.push_back( st3 );
```

冗余元素的产生并不可怕，容器中产生冗余元素之后，可以利用 STL 大侠的“瘦身大法”将这些冗余元素删除，重新恢复苗条的身材。STL 大侠的“瘦身大法”一共有三步。

首先，要对容器中的所有数据进行排序。例如：

```
// 第一步：排序
sort(vecStu.begin(), vecStu.end(), sortByHeight);
```

其次，也是最关键的一步，使用 unique() 算法删除容器中的冗余元素。unique() 算法会调用容器中数据类型的相等运算符 (==) 来判断两个数据元素是否相等，如果这两个元素相等，就删除其中的一个元素，只保留其中的一个。因为这里使用的是自定义的 Student 类型，所以需要先重载这个类的“==”运算符，然后调用 unique() 算法删除其中的冗余元素。

```

// 如果两个 Student 对象的名字和身高都相同，
// 就认为这两个对象完全相同，属于冗余的数据元素
class Student
{
public:
    bool operator == (Student st)
    {
        return m_strName == st.GetName()
            && m_nHeight == st.GetHeight();
    }
// ...
};
// 第二步：删除容器中的冗余元素
vector<Student>::iterator it = unique( vecStu.begin(),
    vecStu.end());

```

使用 `unique()` 算法删除容器中的冗余元素之后，它会返回一个迭代器指向容器中剩下的正确数据元素的位置。但是，在容器的末尾还有一些因为删除元素而遗留下来的多余元素，所以还需要作最后一步的清理工作，将这些多余的元素删除，这样才能真正删除容器中的冗余元素。

```

// 第三步：删除容器末尾遗留的多余元素
vecStu.erase( it, vecStu.end() );

```

从此，身轻如燕！



函数指针、函数对象与 Lambda 表达式

在 C++ 世界中，函数的人缘最广，可以说无处不在，到处都有它的身影：从表达运算过程的普通函数到表示类行为的成员函数，特别是在 STL 算法中，算法的核心逻辑往往是以函数的形式来表达的。在一些通用算法中，通常需要配合一些函数使用，以达到对通用算法进行自定义的效果。比如，一个通用的 `sort()` 排序算法，可以通过向它提供一个排序函数来自定义排序的规则；一个通用的 `find_if()` 查找算法，也可以通过函数对其匹配规则进行自定义。函数极大地增强了通用算法的表达能力，使得通用函数做到了真正通吃天下。

这些能够在通用算法中使用的函数，虽然它们的本质都是函数，但是它们都爱穿马甲，常常变换自己的表现形式：有简单的函数指针，也有复杂的函数对象，更有灵活而优雅的 Lambda 表达式。下面就来看看这些函数的马甲，以免它们换个马甲就不认识了。

11.1 函数指针

顾名思义，函数指针就是指向函数的指针变量。函数指针本身应是一个指针变量，其实质就是内存地址，只不过函数指针变量指向的不是普通的数据变量的内存地址，而是指向一个函数而已。在 C++ 世界中，每个函数都有一个入口地址，该入口地址就是函数指针所指向的内存地址。有了指向函数的指针变量后，就可以用这个函数指针变量调用函数，如同用数据指针变量可以引用它所指向的数据一样，从所指向的内容的引用上，函数指针和普通的数据指针是一致的。

11.1.1 函数指针的声明与赋值

指针的本质就是内存中的某个地址。如果该内存地址中存放的是某个数据，那么这个指针就是常见的数据指针。同样，如果这个内存地址中存放的是某个函数，那么这个指针就是函数指针。

首先，函数指针本身是指针变量，只是该指针变量指向的不是通常所见的整型变量、数组等，它指向的是一个函数。C++ 代码经过编译后，每个函数都有一个入口地址，而函数名就代表这个入口地址。如果某个函数指针指向这个函数，那么该入口地址就是函数指针所指

向的地址。

根据函数指针所指向的函数的不同，需要根据函数的具体声明来定义一个函数指针，其语法格式如下：

函数返回值类型标志符 (指针变量名)(形参列表);

其中，函数返回值类型标志符就是这个指针所要指向的函数的返回值类型。指针变量名就是函数指针变量名，由于“()”的优先级高于“*”，所以指针变量名外的括号必不可少。形参列表表示指针变量指向的函数所带的参数列表。例如，有这样一个函数：

```
// 声明一个函数
void PrintPass( int nScore );
```

如果要声明一个函数指针指向这个函数，则可以使用下面的代码：

```
// 定义函数指针
void (*pPrintFunc)( int nScore );
```

这样，就声明了一个可以指向 PrintPass()函数的函数指针 pPrintFunc。当声明函数指针时，参数列表中的形式参数名可有可无，以上代码也可以写成下面这种简化的形式：

```
// 省略形式参数的函数指针声明
void (*pPrintFunc)( int );
```

当函数的形式参数比较多时，通常省略形式参数名，让函数指针的声明更加简洁。

如果要定义多个同一类型的指针，还可以使用 typedef 关键字定义一种新的函数指针的数据类型，用这种新的数据类型来定义函数指针。例如：

```
// 定义一种新的函数指针的数据类型
typedef void (* PRINTFUNC )(int);
// 使用新的数据类型定义函数指针
PRINTFUNC pFuncFailed;
PRINTFUNC pFuncPass;
```

这里，就定义了一种新的函数指针类型 PRINTFUNC，它表示这种函数指针类型可以指向一个参数为 int、返回值为 void 的函数。使用这种新的函数指针类型，可以连续定义多个函数指针，指向多个相同类型的函数。

从这里可以注意到，函数指针的声明跟它要指向函数的声明非常相似，函数的返回值和参数列表都相同，只是将函数名换成了指针的形式。这也隐含了一个实质：函数名就是指向函数的指针。完成函数指针的声明后，就可以用函数名给函数指针赋值，让它指向这个函数。

```
// 用函数名给函数指针赋值
pPrintFunc = PrintPass;
```

这种使用 typedef 定义函数指针类型的方式虽然方便，但是定义的时候过于烦琐。C++语

言真是体贴入微，早就想到了这一点，它提供了一个 auto 关键字。可以利用这个 auto 关键字作为函数指针的数据类型来声明一个函数指针，至于这种类型的具体定义就留给编译器自己去推断吧，因为它最擅长干这些活，而程序员的脑子就用来思考问题吧。利用 auto 关键字，可以这样定义函数指针类型：

```
// 利用 auto 关键字定义函数指针类型
// 编译器会在变量赋值的时候，自动推断函数指针的具体说明
auto pPrintFunc = PrintPass;
```

11.1.2 用函数指针调用函数

有了指向数据的数据指针之后，可以通过数据指针访问它所指向的数据。同理，有了指向函数的函数指针，也可以利用这个函数指针调用函数，就像直接调用这个函数一样。例如：

```
// 通过函数指针调用函数
(*pPrintFunc)( 75 );
```

这里，就利用函数指针实现了函数调用，因为函数指针 pPrintFunc 指向的是 PrintPass() 函数，它实际上相当于如下的函数调用：

```
// 实际的函数调用
PrintPass( 75 );
```

既然使用函数指针调用函数跟普通函数调用没有什么差别，那么何必使用函数指针来调用函数多此一举呢？直接使用普通的函数调用就好了，简单又方便！

没错，使用普通的函数调用是既简单又方便，但是不够灵活！指针的灵魂就是它的灵活性！下面来看一个实际的例子，看看函数指针是如何利用它的灵活性让程序轻舞飞扬的。在一个成绩管理程序中，需要根据学生成绩打印出不同的消息。使用普通的函数调用，可以这样实现：

```
// 消息打印函数，根据不同的分数打印不同的消息
void PrintPass( int nScore )
{
    cout<<"分数是："<<nScore<<" 恭喜你通过考试！"<<endl;
}
void PrintFailed( int nScore )
{
    cout<<"分数是："<<nScore<<" 很抱歉，没有通过考试！"<<endl;
}
void PrintExcellent( int nScore )
{
    cout<<"分数是："<<nScore<<" 哇，你是天才！"<<endl;
}

// 主函数
int _tmain(int argc, _TCHAR* argv[])
```

```

{
    int nScore = 22;
    // 根据分数进行判断
    // 不同的分数调用不同的函数，打印不同的消息
    if( nScore < 60 )        // 不及格
    {
        PrintFailed( nScore );
    }
    else if( nScore >= 60 && nScore < 100 )    // 及格
    {
        PrintPass( nScore );
    }
    else        // 优秀
    {
        PrintExcellent( nScore );
    }

    return 0;
}

```

虽然这段代码能够实现根据不同分数打印不同消息的功能，但是整段代码显得非常笨重。如果打印消息的函数增多，switch 语句中将有越来越多的函数调用，代码也会越来越臃肿。使用函数指针，则可以为这段代码瘦身，让它轻舞飞扬起来。

```

// 使用函数指针改写后的瘦身版主函数
int _tmain(int argc, _TCHAR* argv[])
{
    int nScore = 22;

    // 定义函数指针
    void (*pPrintFunc)(int);
    // 根据分数进行判断
    // 不同的分数，使用不同的函数对函数指针进行赋值
    if( nScore < 60 )
    {
        pPrintFunc = PrintFailed;
    }
    else if( nScore >= 60 && nScore < 100 )
    {
        pPrintFunc = PrintPass;
    }
    else
    {
        pPrintFunc = PrintExcellent;
    }

    // 最后，以统一的函数指针的形式调用函数
    // 因为函数指针被不同的函数入口地址赋值，从而实现了不同函数的调用
    (*pPrintFunc)( nScore );

    return 0;
}

```

在改写后的瘦身版主函数中，首先定义了一个可以指向打印函数的函数指针 pPrintFunc，然后根据分数的不同，将不同的打印函数入口地址赋值给这个函数指针；最后以统一的函数指针的形式调用打印函数。这样，就实现了以同一个函数指针、以统一的方式实现了不同函数的调用，达到了给程序瘦身的目的。同时，一致的函数调用方式也更容易理解，具有更高的可读性。

11.1.3 用函数指针实现回调函数

除了可以使用函数指针简化函数的调用之外，函数指针更大的用途在于它可以作为函数参数传递给某个函数，从而实现函数的回调。所谓函数的回调，就是在某个函数中，通过函数指针调用另外一个函数，而这个函数指针，大多数情况下是通过函数参数传递进来的。如果把函数的指针作为参数传递给另一个函数，那么当这个函数指针用于调用它所指向的函数时，就说这个函数是回调函数。

函数指针已经够复杂了，为什么还要使用它来作为函数参数进行函数的回调？程序简单一点不是更好吗？

回调函数的使用，正是为了让程序变得更简单。因为回调函数可以把主调函数与被调函数分开，主调函数不必关心谁是被调函数，所有它需要知道的只是存在一个具有某种特定原型、某些限制条件的被调函数。这就像在主调函数中留下了一个插口，它定义了插口的规则，也就是函数的返回值和具体的参数。而被调函数就是插头，可以把任何符合这个插口规则的插头插入这个插口中，从而实现整个主调函数。也可以通过改变插入的插头来改变主调函数的功能，从而改变主调函数的实现。使用回调函数可以实现同一个算法框架、不同的算法实现，最终达到算法的通用。

还是回到上面的例子，看看如何使用回调函数让程序更具灵活性、更加通用。假设现在要求在每条打印消息的前后还要打印一些符号作为装饰，为了完成这项任务，可以定义一个统一的打印消息函数。

```
// 定义函数指针类型
typedef void (* PRINTFUNC )(int);
// 统一的打印消息函数
void PrintMessage( int nSocre , PRINTFUNC pFunc )
{
    cout<<"======"<<endl;
    // 通过函数指针回调函数
    (*pFunc)(nSocre);
    cout<<"++++++"<<endl;
}
// 主函数
int _tmain(int argc, _TCHAR* argv[])
{
```

```

    int nScore = 22;

    PRINTFUNC pFunc;
    // 根据不同分数给 pFunc 赋值
    // ...

    // 使用不同函数指针作为参数调用 PrintMessage() 函数
    PrintMessage( nScore, pFunc );

    return 0;
}

```

在这里，实际上是通过 PrintMessage() 函数定义了一个通用算法框架：首先打印页眉；然后通过函数指针回调函数，打印具体的消息；最后打印页脚。PrintMessage() 函数只完成最基本的页眉、页脚的打印，至于具体的消息，则留给回调函数负责，这就像留下一个插口，等待某个具体的回调函数插头的插入。在主函数中，通过给 PrintMessage() 函数传递不同的打印函数的指针，就如同将某个插头插入 PrintMessage() 函数所留下的插口中。不同函数指针插头的插入，可以改变 PrintMessage() 函数中的回调函数，进而改变 PrintMessage() 函数的行为，以达到对其行为进行自定义的效果。回调函数与插头理论的关系如图 11-1 所示。

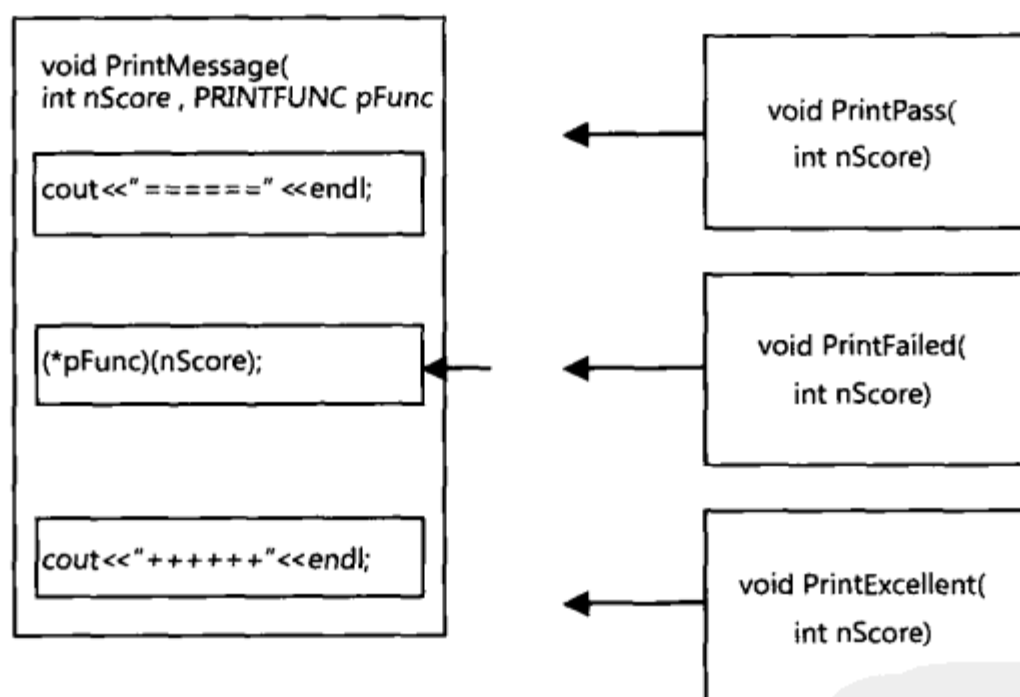


图 11-1 回调函数与插头理论

通过回调函数改变一个函数的行为、对其行为进行自定义的特性被广泛应用在通用算法的设计中。假设有这样一种情况，要编写一个库，它提供了某些排序算法的实现，如冒泡排序、快速排序等，为了使库更加通用，不想在函数中嵌入排序逻辑，而让使用者来实现具体的排序逻辑，或者让库用于多种数据类型。这时，回调函数就可以派上用场了。可以在通用算法中定义好算法的框架，至于其中核心的算法逻辑，则留待回调函数去完成。用户可以通过不同的回调函数，轻松简单地实现各种算法，对算法进行自定义。

11.1.4 将函数指针应用到 STL 算法中

就像上文中介绍的回调函数一样，可以通过给一些 STL 算法提供一个函数指针，对其中的核心业务逻辑进行自定义，以最终达到让 STL 中的通用算法也同样能够满足个性化需求的目的。例如，容器中保存的是 Student 对象，可以为 STL 中的 count_if() 提供一个函数指针，让它统计所有身高大于 170 的学生人数。

```
// 利用函数定义统计的规则
bool countHeight( Student st )
{
    // 如果身高大于 170，则统计在内
    return st.GetHeight() > 170;
}

// 将统计函数的指针 countHeight 应用到 count_if() 算法中
// 这样 count_if() 算法将调用 countHeight() 函数进行统计
int nCount = count_if(vecStu.begin(), vecStu.end(), countHeight );
cout<<"身高大于 170 的学生有："<<nCount<<endl;
```

在这段函数中，首先定义了一个函数 countHeight()，并在其中表述了自己的统计规则。然后，将这个函数指针，也就是函数名 countHeight 作为参数调用 STL 中的 count_if() 算法。当 count_if() 算法对容器中的数据进行统计时，会调用 countHeight() 函数来判断当前数据是否需要统计在内。换句话说，以函数指针的形式作为参数传递给 count_if() 算法的 countHeight() 函数成了一个回调函数，它会在 count_if() 算法中被调用。这样，可以在 countHeight() 函数中实现自己的统计规则，从而让 count_if() 算法按照规则进行统计，最终实现利用函数指针对 STL 算法进行自定义的目的。

从以上这段函数中注意到，countHeight() 函数中的标准身高已经固定下来，这就让整个统计算法失去了灵活性。如果要统计大于另外一个身高的人数，则不得不重新编写另外一个统计规则函数。为了让这个统计算法具有更大的灵活性，则要对统计规则函数 countHeight() 进行改写，将统计标准也作为函数的参数，在调用时给定统计标准，让这个统计算法更加灵活。

```
// 将统计标准也作为参数，重新定义统计规则函数
bool countHeight( int nHeight, Student st )
{
    // 如果身高大于标准身高，则统计在内
    return st.GetHeight() > nHeight;
}
```

当调用 count_if() 算法进行统计时，就可以自由地指定标准身高，灵活地完成统计。

```
// 定义标准身高
int nStandardHeight = 170;
int nCount = count_if(vecStu.begin(), vecStu.end(),
```



```

        bind1st( ptr_fun(countHeight), nStandardHeight) );
    cout<<"身高大于"<<nStandardHeight<<"的学生有: "<<nCount<<endl;

```

在这里，首先使用 `ptr_fun()` 函数将一个普通函数指针转换为一个函数对象，给函数换了一件马甲；然后用 `bind1st()` 函数将整个函数对象的第一个参数绑定为 `nStandardHeight`，而函数对象的第二个参数就是容器中的 `Student` 对象。这样，`count_if()` 算法就会以 `nStandardHeight` 和容器中的 `Student` 对象作为参数调用 `countHeight()` 这个重新定义的统计规则函数，实现统计规则和统计标准的完全自定义。

除了可以在 STL 算法中使用指向普通函数的函数指针外，还可以在算法中使用指向某个类的成员函数的函数指针。例如，STL 中的 `for_each()` 算法就经常使用指向类的成员函数的指针来调用容器中所保存的对象的成员函数，而 `count_if()` 和 `find_if()` 等算法也常常使用成员函数指针调用容器所保存的对象的成员函数，以此来判断这个对象是否符合某个条件。例如，要在保存 `Student` 对象的容器 `vecStu` 中找到某个名字叫“ChenLiangqiao”的 `Student` 对象，就可以直接使用成员函数指针调用 `Student` 类的成员函数 `isnamed()`，以此来判断当前对象是否就是我们所要找的人。

```

// 学生类
class Student
{
public:
    // 公有的成员函数
    // 判断学生的名字是否跟参数相同
    bool isnamed(string strName)
    {
        return strName == m_strName;
    }
private:
    string m_strName;
};

```

通过定义 `isnamed()` 成员函数，`Student` 类本身具备判断能力后，在 `find_if()` 算法中就可以直接使用指向 `isnamed()` 这个成员函数的函数指针来判断当前对象是否就是我们要找的对象。

```

// 要查找的名字
string strFindName = "Jiawei";
// 使用成员函数指针判断容器中的对象是否符合查找条件
auto it = find_if(vecStu.begin(), vecStu.end(),
                 bind2nd( mem_fun_ref(&Student::isnamed), strFindName ) );
// 如果找到符合条件的对象，则输出查找结果
if( it != vecStu.end() )
    cout<<"找到了名字为"<<strFindName<<"的 Student 对象"<<endl;

```

在这段代码中，首先使用“&”运算符获得 `Student` 类的成员函数 `isnamed()` 的地址，也就是获得了指向这个成员函数的函数指针；然后使用 `mem_fun_ref()` 函数将这个函数指针构造成为一个函数对象，如果容器中保存的是指向对象的指针，就应该使用 `mem_fun()` 函数来完成

这一任务。因为这个成员函数需要一个参数，所以更进一步地，使用 `bind2nd()` 函数绑定其第二个参数为 `strFindName` 来作为查找条件。因为这是一个类的成员函数指针，所以容器中的对象会作为其默认隐含的第一个参数。有了指向成员函数的函数指针，在使用这些对象本身所具备的功能时更加简单。

函数指针配合 STL 算法的应用，将 STL 算法的通用性发挥到了极致。

11.2 函数对象

函数指针虽然能够增加 STL 算法的通用性，但是在实际应用中我们发现函数指针有一个致命的缺点：函数患有严重的健忘症，它无法记住它上一次调用时的状态。它只是认认真真地完成一个运算过程，至于上一次函数调用时的一些状态数据，它一点记忆都没有。这就让它无法应用在那些需要对每次调用状态进行维护的场景。例如，无法应用单独的一个函数来计算容器中的所有数据的和，因为它无法得知上一次累加的结果是多少。这时，就该函数换上另外一件马甲——函数对象了。

11.2.1 定义一个函数对象

所谓函数对象，就是定义了函数调用操作符（function-call operator），即 `operator()` 的普通类的对象。在重载的函数调用操作符中，可以实现函数的所有功能。同时，因为类具有属性，可以将每次函数调用的状态数据保存到它的属性中，这样函数对象就不会像函数指针那样失忆了，从而可以应用在更广的范围内。

跟定义一个普通函数差不多，要定义一个函数对象也非常简单。首先，需要定义一个普通的类并重载它的函数调用操作符“`operator()`”。例如，可以定义一个函数对象类来实现比较两个数大小的功能。

```
// 模板函数对象
template <typename T>
class mymax
{
public:
    // 重载"()"操作符，在这个操作符中实现具体功能。
    // 这使得一个普通类成为一个函数对象。
    T operator ()(T a, T b)
    {
        return a > b ? a : b;
    }
};
```

在重载“`()`”操作符的语句中，第一个圆括弧总是空的，因为它代表重载的操作符名，

第二个圆括弧中就是这个函数对象的参数列表，它跟一个普通函数的参数列表相同。一般在重载操作符时，参数数量是固定的，而重载“()”操作符时有所不同，它可以有任意多个参数。

定义好函数对象类之后，就可以开始使用函数对象类创建相应的函数对象，并利用这些函数对象来进行具体的计算。既然函数对象是一个具体的类的实例变量，那么它既可以单独使用，也可以像函数指针一样被当成参数传递给其他函数，并在其他函数中使用。

```
// 利用函数对象定义函数模板
// 执行具体的比较操作
template <typename T>
T compare(T a, T b, mymax<T>& op)
{
    // 利用传递进来的函数对象进行具体的比较操作
    return op(a, b);
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义一个 int 类型的函数对象，它可以比较两个 int 类型的数
    // 并返回较大的数
    mymax<int> intmax;
    // 直接使用函数对象完成比较操作
    int nMax = intmax(3, 4);
    // 将函数对象作为参数传递给其他函数，
    // 在其他函数中完成对函数对象的使用
    nMax = compare(2, 3, intmax);

    return 0;
}
```

在这段函数中，实现了函数对象的直接使用和作为参数传递给其他函数的间接使用。不管使用哪种方式，函数对象的调用方式都是相同的，都是使用“()”操作符加上相应的参数完成对函数对象的调用，这跟普通的函数调用没有什么差别。简单地讲，函数对象就是常规的类对象，可以采用标准的函数调用语法来调用它的函数调用操作符“operator()”成员函数。编译时，编译器会将函数对象的调用语句

```
intmax(3, 4);
```

转化为

```
intmax.operator()(3, 4);
```

从本质上讲，函数对象的调用也就是它调用了这个函数对象的一个特殊成员函数——operator()。

通常来说，函数对象不定义构造函数和析构函数，因此，在创建和销毁过程中就不会发生任何问题。同时，重载的“()”是一个内联函数，编译器能内联重载的操作符代码，从而

避免了与函数调用相关的运行问题。

11.2.2 利用函数对象记住状态数据

函数只是用来表达一个运算的过程，它无法记住运算过程中的一些状态数据。函数就像一个漏斗，数据可以从这个漏洞中流过，发生某些变化，但是这个漏斗什么都不会留下。有时候是需要记住函数执行过程中的状态数据的。例如，用一个函数统计容器中所有 Student 对象的身高，就需要在上次累加结果这个状态数据的基础上进行累加，将所有身高都累加到一起。如果函数无法记住这个状态数据，每次就从零开始统计，这样始终都无法完成身高的统计，成了函数永远无法完成的任务。

好在函数对象可以弥补函数的这个缺陷，利用函数对象自身的成员属性可以记住函数在每次执行过程中的状态数据，以找回失去的记忆。例如，可以利用函数对象来替函数完成它那永远也无法完成的任务。

```
// 定义一个函数对象类
// 用于统计容器中所有 Student 对象的平均身高
class AverageHeight
{
public:
    // 构造函数，对类的成员属性作合理的初始化
    AverageHeight()
        : m_nCount(0), m_nTotalHeight(0) {};

    // 重载函数调用操作符“()”，
    // 在其中完成统计的功能
    void operator () ( Student st )
    {
        // 将当前对象的身高累加到总身高中
        // 这里的 m_nTotalHeight 记录了上次累加的结果，
        // 这就是函数那失去的记忆
        m_nTotalHeight += st.GetHeight();
        // 增加已经统计过的 Student 对象的数目
        ++m_nCount;
    }

    // 接口函数，获得所有统计过的 Student 对象的平均身高
    float GetAverageHeight()
    {
        if ( 0 != m_nCount )
            return (float)GetTotal()/GetCount();
    }

    // 获得函数对象类的各个成员属性
    int GetCount()
    {
        return m_nCount;
    }
}
```

```

    int GetTotal()
    {
        return m_nTotalHeight;
    }

    // 函数对象类的属性,
    // 用来保存函数执行过程中的状态数据
private:
    int m_nCount;           // 记录已经统计过的对象的数目
    int m_nTotalHeight;     // 记录已经统计过的身高总和
};

```

为了让函数对象完成身高统计的功能，我们在函数对象类中添加了两个属性来记录函数调用过程中的状态数据：m_nCount 用于记录已经统计过的对象的数目，也就是当前已经统计了多少 Student 对象的身高数据；m_nTotalHeight 则用来记录已经统计过的身高总和，这样针对每个对象的统计都会累加到这个身高总和成员属性上，函数也就不会再失忆了。

现在有了这个可以找回记忆的函数对象类，就可以创建该类的函数对象来完成统计身高的任务了。

```

// 创建函数对象
AverageHeight ah;
// 将函数对象应用到 STL 算法中
ah = for_each( vecStu.begin(), vecStu.end(), ah);
// 从函数对象中获取它的记忆作为结果输出
cout<<ah.GetCount()<<"个学生的平均身高是: "
    <<ah.GetAverageHeight()<<endl;

```

在这里，创建了一个函数对象 ah 并将它应用到 for_each() 算法中，for_each() 算法会以容器中的每个 Student 对象为参数来对这个函数对象的 “()” 操作符进行调用。这时函数对象自然就会将每个 Student 对象的身高累加到它自己的 m_nTotalHeight 属性上，同时它还会记录已经统计过的对象数目。最后，for_each() 算法会将完成统计后的函数对象作为结果返回，而这时的函数对象 ah 已经是一个保存了统计结果的函数对象了。通过函数对象提供的接口函数，可以轻松地获得统计结果并进行输出。更直接地，可以在函数对象类中定义一个类型转换函数，将函数对象直接转换为所需要的目标结果。例如：

```

class AverageHeight
{
    // ...
    // 定义类型转换函数
    // 将函数对象转换为 float 类型，直接返回计算结果
    operator float ()
    {
        return GetAverageHeight();
    }
};

```

现在，就可以直接从 for_each() 算法中获得计算结果了。

```
// 直接从 for_each() 算法获得计算结果
flott fAH = for_each( vecStu.begin(), vecStu.end(), ah() );
```

穿上函数对象这个马甲，函数不再仅仅是一个过程，过去了就过去了。从此，它找回了自己失去的记忆。

知道更多：STL 中的函数对象

为了减少定义函数对象类的工作，STL 中预定义了许多常用的函数对象，这些对象包括：

- 算术操作

这类对象可以用于常见的算术运算，例如，加（plus）、减（minus）、乘（multiplies）、除（divides）、取余（modules）和取负（negate）等。

- 比较操作

这类对象可以用于进行数据的比较，例如，等于（equal_to）、不等于（not_equal_to）、大于（greater）、小于（less）、大于等于（greater_equal）、小于等于（less_equal）。

- 逻辑操作

这类对象可以用于逻辑运算，例如，逻辑与（logical_and）、逻辑或（logical_or）、逻辑非（logical_not）。

可以直接在程序中应用这些常用的函数对象来完成相应的操作。例如，可以使用比较操作对象来统计容器中大于某个值的数据个数。

```
// 统计容器中大于 60 的数据个数
int nCount = count_if( vecScore.begin(),
    vecScore.end(), bind2nd( greater<int>(), 60) );
```

11.3 用 Lambda 表达式编写更简单的函数

在 11.1 节中见识过，函数换个马甲就可以摇身变成函数指针或者函数对象。现在的函数则更厉害了，可以连名字都没有就变成匿名函数（anonymous function）。所谓匿名函数，就是这个函数只有函数体，而没有函数名。Lambda 表达式就是实现匿名函数的一种编程技巧，它为编写匿名函数提供了简明的函数式句法。

11.3.1 最简单直接的函数表达形式：Lambda 表达式

虽然函数对象可以解决函数指针无法保存状态数据的问题，但是，如果在程序中大量使用函数对象，则会使代码显得非常累赘而冗长。一方面，针对每个算法，都要创建一个完整

的函数对象类，包括构造函数和重载的“()”操作符，即使它只是一个简单的求和算法。如果程序中有很多算法使用函数对象实现，就会导致程序中类的数目增多，类出现膨胀的情况。编写这些函数对象类也相当枯燥而烦琐。另一方面，因为函数定义的地方并不在它使用的地方，所以它会打断代码的流程，使代码不够流畅自然。如果想在使用函数对象的地方查看它的定义，则不得不跋山涉水、翻山越岭，在多个代码文件之间跳转，这同样会影响开发的效率。更严重的是，由于无法控制函数对象类的使用，其他程序员可能误用某个函数对象类而出现严重的算法错误。正是这些原因，如果在程序中大量使用函数对象，也会使代码毫无“优雅”可言。

为了解决这些问题，C++为函数精心缝制了一件新马甲：Lambda 表达式。在作用上，Lambda 表达式类似于函数指针和函数对象，它同样可以方便地应用于 STL 中，对算法进行自定义。在使用形式上，Lambda 表达式可以在使用函数的地方对其进行定义，使整个代码更加自然流畅。同时，可以在 Lambda 表达式中访问 Lambda 表达式之外的数据，解决函数执行过程中状态数据的保存问题，为容器的操作带来很大的便利。Lambda 表达式很好地兼顾了函数指针和函数对象的优点，而没有它们的缺点。相对于函数指针或函数对象复杂的语法形式，Lambda 表达式使用非常简单的语法就可以实现同样的功能，降低 Lambda 表达式的使用难度，避免使用复杂的函数对象或函数指针可能带来的错误。下面还是先来看看这件新马甲如何“优雅”程序的：

```
// 定义变量，用于保存函数的状态数据
int nTotalHeight = 0;
int nCount = 0;
// 在 for_each() 算法中使用 Lambda 表达式，统计身高
for_each( vecStu.begin(), vecStu.end(),
    [&](Student st) // Lambda 表达式
    {
        nTotalHeight += st.GetHeight();
        ++nCount;
    });
cout<<nCount<<"个学生的平均身高是："
    <<(float)nTotalHeight/nCount<<endl;
```

短短的几行代码，就完成了原来需要一个函数对象类才能完成的工作。除了鼓掌，想不到还能用别的什么方式来表达赞叹之情。

这是一段 Lambda 表达式与 for_each() 算法的绝妙配合。在这段代码中，使用一对中括号“[]”来表示 Lambda 表达式的开始，其后的“(Student st)”就是这个表达式的参数。当 for_each() 算法循环遍历容器中的每一个元素时，它会将容器中的元素作为参数调用 Lambda 表达式。在 Lambda 表达式内部，可以获得容器中 Student 对象的身高属性，并将它累加到 nTotalHeight 以实现函数执行过程中状态数据的保存。最后，nTotalHeight 和 nCount 都保存需要的统计数据，进行简单的计算并将结果输出就完成了整个统计工作。

使用 Lambda 表达式，不需要创建额外的函数对象，也无须中断代码流程，一切都显得那么流畅自然。

11.3.2 Lambda 表达式的语法规则

从上面的示例代码可以看到，Lambda 表达式的定义很简单，它的语法格式如下：

```
[变量使用说明符号](参数列表) -> 返回值数据类型
{
    // 函数体
}
```

其中，中括号“[]”表示 Lambda 表达式的开始，用来告诉编译器接下来的代码就是 Lambda 表达式。在中括号中，可以使用 Lambda 表达式定义当前作用域中的变量方式。如果是传值（复制）方式，则使用“[=]”表示，它将使得 Lambda 表达式以只读的方式访问当前作用域内的变量。换句话说，就是只能在 Lambda 表达式中读取这些变量的值，而不能修改这些变量的值。如果试图在 Lambda 表达式中对这些变量进行修改，将产生一个编译错误。如果中括号留空，默认情况下也表示以传值方式使用当前作用域内的变量。例如：

```
vector<int> v;
// 向容器中添加数据...
int nAdd = 3;
for_each(v.begin(), v.end(),
    [=](int x)           // 以传值的方式使用当前作用域的变量
{
    x += nAdd;           // 只读访问当前作用域的变量
    cout<<x<<endl;
});
```

如果还想在 Lambda 表达式中修改当前作用域内的变量，就需要使用传引用的方式来定义 Lambda 表达式。这时可以使用“[&]”代替“[=]”来说明 Lambda 表达式将以传引用的方式使用当前作用域内的变量。因为 Lambda 表达式的变量都是外部同名变量的引用，所以在 Lambda 表达式中对这些变量的修改将直接影响到当前作用域中变量本身。例如：

```
int nTotal = 0;
for_each(v.begin(), v.end(),
    [&](int x)           // 以传引用的方式使用当前作用域的变量
{
    nTotal += x;         // 修改变量的值
});
cout<<"容器中数据的总和是："<<nTotal<<endl;
```

这里，使用了一个定义在 Lambda 表达式之外的局部变量 nTotal 来保存 vector 容器中所有数据的总和，实现了从 Lambda 表达式向外部传递数据的功能。

如果需要与 Lambda 表达式传递多个数据，进行更加复杂的数据交换，那么可以在“[]”

的第一位设置一个默认的传递方式，然后再分别指定各个变量的传递方式。例如，想在默认情况下使用传引用的方式，但是其中一个变量使用传值的方式，可以这样定义 Lambda 表达式：

```
int nAdd = 3;
int nTotal = 0;
for_each(v.begin(), v.end(),
    [&, nAdd](int x)           // 默认采用传引用访问，nAdd 使用传值访问
    {
        nTotal += (x * nAdd);
    });
cout<<"容器中的数据乘以"<<nAdd<<"之后总和是"<<nTotal<<endl;
```

在 Lambda 表达式的传数据声明之后，就是 Lambda 表达式的参数列表。它的参数列表跟普通函数的参数列表类似，主要用于接收 STL 算法传递进来的数据。

通常来说，Lambda 表达式没有返回值，这时可以省略返回值类型的定义。如果某些算法需要 Lambda 表达式有返回值，则可以使用“->”符号来定义它的返回值类型。例如：

```
// 统计容器中偶数的个数
int nEven = count_if(v.begin(), v.end(),
    [=](int x) -> bool        // 定义 Lambda 表达式的返回值类型为 bool 类型
    {
        // 判断是否是偶数，返回 bool 类型
        return x % 2 == 0;
    });
```

这里，Lambda 表达式的返回值是 bool 类型，表示一个数是否是整数。而 count_if() 算法则根据这个返回值进行统计，计算容器中偶数的个数。

通过 Lambda 表达式灵活的定义方式，可以实现各种各样的 Lambda 表达式与 STL 算法之间的数据交换，从而使得 Lambda 表达式与 STL 算法配合得天衣无缝。

11.3.3 Lambda 表达式的复用

Lambda 表达式有这么好的函数马甲，如果只能穿一次，只能在一个 STL 算法中使用，那多么可惜啊！C++想到了这一点。实际上，跟函数对象可以重复多次使用一样，也可以把 Lambda 表达式储存起来应用到多个 STL 算法中，实现 Lambda 表达式的复用。

```
// 定义 vector 容器
vector<int> intvector;
// 向 vector 中添加数据

// 定义 list 容器
list<int> intlist;
// 向 list 中添加数据

// 定义一个可以输出整数的 Lambda 表达式
```

```

auto show = [] ( int x )
{
    cout<<x<<endl;
};
// 在 vector 容器上应用这个 Lambda 表达式
for_each(intvector.begin(),intvector.end(),
        show);
// 在 list 容器上重复应用这个 Lambda 表达式
for_each(intlist.begin(),intlist.end(),
        show);

```

换句话说，无法得到一个还没有实现的 Lambda 表达式，Lambda 表达式实际上也可以看成是一个依赖于实现的函数对象，所以，甚至可以把保存起来的 Lambda 表达式作为参数传递给函数。例如：

```

// 可以接受函数对象为参数的函数
void PrintMsg( tr1::function<void (int)> func )
{
    vector<int> v;
    v.push_back(3);
    for_each(v.begin(), v.end(),
            func); // 使用传递进来的 Lambda 表达式
}

// 使用 Lambda 表达式作为参数调用 PrintMsg() 函数
PrintMsg( show );

```

好东西要和好朋友一起分享，Lambda 表达式有这么好的马甲，当然要重复使用，大家一起分享啦！



C++世界的几件新鲜事

“卖报啦卖报啦——”

“C++世界的新标准 C++0x 新鲜出炉，大家都来看看 C++世界的新鲜事啊！”

“右值引用心狠手辣想榨干 C++的性能，智能指针聪明伶俐想解决 C++内存管理的问题，PPL 正在做慈善、大量派发免费午餐啊……”

12.1 用右值引用榨干 C++的性能

在性能方面，除了汇编语言和 C 语言外，C++大概是目前世界上公认的性能最高的程序设计语言了。但是，C++标准委员会的那帮“老头们”并不就此感到满足，在 C++的最新标准 C++0x 中，它们心狠手辣地添加了右值引用这一新特性，想要榨干 C++身上的最后一点性能。

12.1.1 什么是右值

通常我们接触最多的是数据变量，很少听说右值。那么，右值到底是个什么东西呢？右值引用又是什么呢？

在 C++语言中，按照能否放在赋值操作符“=”的左边或者右边，数值可以分成左值或者右值。通常，我们接触最多的是放在等号左边的左值，它们既能够被赋值，也能够对其他左值赋值，例如数据变量等。但是，在 C++语言中还有另外一类数值，它们只能放在等号右边，只能用于对左值赋值，这样的数值称为右值。右值通常是一些数值常量、临时变量或者无名变量等，例如一个函数的返回值就是右值。而右值引用，顾名思义，也就是针对这些右值的引用了。在 C++语言中，可以通过运算符“&&”来声明一个右值引用，而原先在 C++语言中使用运算符“&”声明的引用现在称为左值引用。例如：

```
// 返回一个临时的 int 类型的数值
// 这个函数的返回值就是右值
int CreateInt(int nInt)
{
    return int(nInt);
}

// 定义一个 int 类型的变量，这个变量就是左值
```

```
int nInt;
// 定义一个左值引用，将它指向一个左值
int& lrefInt = nInt;
// 定义一个右值引用，将它指向一个右值
int&& rrefInt = CreateInt(4);
```

左值引用和右值引用的表现行为基本一致，它们唯一的差别就是右值引用只能绑定到一个右值（临时对象）而不能绑定到一个左值。同样，左值引用只能绑定到一个左值而不能绑定到一个右值。例如：

```
// 左值引用不可以绑定到右值，否则将产生编译错误
int& lrefInt = CreateInt(4);
// 右值引用也不可以绑定到左值
int&& rrefInt = nInt;
```

在这里，尝试将函数的返回值绑定到一个左值引用，会产生一个编译错误。在第 2 行代码中，尝试将一个变量绑定到一个右值引用，同样会产生一个编译错误。简单来讲，就是左值引用只能绑定到左值，右值引用只能绑定到右值。

绑定完成之后，左值引用和右值引用的使用就没有区别了，它们都可以当成普通数据变量进行左右值的赋值操作。例如：

```
// 对右值引用赋值
rrefInt = 1;
// 利用右值引用对左值引用赋值
lrefInt = rrefInt;
```

12.1.2 右值引用在函数返回值上的应用

实际上，右值就是一些无名的数据变量。例如，函数的返回值没有名字就是右值，数值常量没有名字也说它是右值。当使用右值对变量进行赋值时，通常是把右值复制到等号左边的变量，等赋值完成后，不但不保留右值，而且还要释放掉资源。在没有右值引用的时代，往往把右值当成不再有用的废物释放掉。例如，从函数中返回一个临时变量，在函数返回后，这个临时变量就被当成废物释放掉。而右值引用的引入，可以将一个引用直接指向这个右值，从而实现废物的再利用。例如，从函数中返回一个临时对象：

```
// 利用函数创建并返回一个类的对象
MemoryBlock GreateBlock( size_t nSize )
{
    return MemoryBlock( nSize );
}
// 利用函数返回值对变量进行赋值
MemoryBlock block = GreateBlock(703);
```

如果不使用右值引用，则整个函数返回与赋值将是一个漫长的过程。首先，在函数中，需要创建临时的 MemoryBlock 对象；然后创建目标 block 对象；接着通过函数返回临时对象

并将其复制到 MemoryBlock 对象；最后，函数中的临时对象不再有用时，会被当成废物清理掉。整个过程进行了同一个对象的多次创建、复制与销毁动作，费时又费力。

如果使用右值引用，整个过程就简单多了。首先，在函数中创建临时对象 MemoryBlock；然后当创建目标对象 block 时，编译器会检测到对对象进行赋值的是一个右值，这样编译器就直接使用函数返回的指向临时对象这个右值的引用完成 block 对象的创建。这里可以看到，block 对象的创建过程并不是通常认为的申请内存、初始化对象属性的过程，而是利用它的右值引用构造函数，直接将整个对象指向右值引用所绑定的对象，即在函数中已经创建的临时对象，再将临时对象移动到目标对象而完成目标对象的创建。通过右值引用，将目标对象直接指向一个右值，省略了中间环节中临时对象的创建、销毁及复制过程，实现了右值的废物再利用，低碳又环保，整个过程如图 12-1 所示。

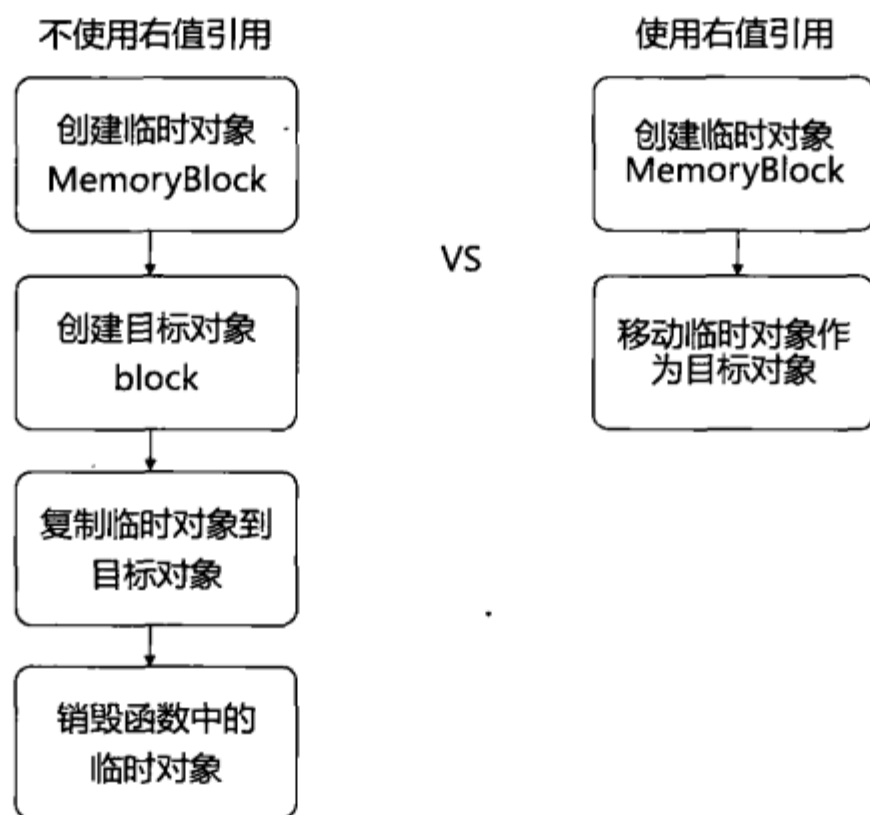


图 12-1 右值引用简化函数返回过程的流程图

12.1.3 STL 算法中被浪费的右值

右值引用除了可以充分利用函数返回值来提高性能外，它还可以大量地应用在 STL 的算法中，以提高通用算法的性能，特别是一些涉及对象的移动复制的算法。考虑一个简单的数据交换的小算法，从中可以体会右值引用是如何提高 STL 算法的性能的。例如，现在有一个用于管理内存的 MemoryBlock 类，它占有一定的内存资源，并拥有一个表示内存资源长度的成员变量 length。

```
// 一个管理内存资源的类
class MemoryBlock
{
```



```

public:
    // 构造函数, 申请内存资源
    MemoryBlock(size_t length)
        : _length(length), _data(new int[length])
        // 初始化成员变量, 并申请内存资源
    {
        cout << "创建对象, 长度 = "
              << _length << ", 申请资源" << endl;
    }

    // 析构函数, 释放内存资源
    ~MemoryBlock()
    {
        cout << "销毁对象, 长度 = "
              << _length << ", ";

        if (_data != NULL)
        {
            cout << "释放资源";
            delete[] _data;    // 释放资源
        }

        cout << endl;
    }

    // 普通的复制构造函数, 接受一个左值引用为参数
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length), _data(new int[other._length])
    {
        cout << "复制构造函数 长度 = "
              << other._length << ", 申请资源并复制资源到目标资源" << endl;

        // 复制源对象的内存资源到目标对象的内存资源
        copy(other._data, other._data + _length, _data);
    }

    // 普通的赋值操作符 "=", 接受一个左值引用为参数
    MemoryBlock& operator=(const MemoryBlock& other)
    {
        if (this != &other)
        {
            cout << "赋值操作符. 长度 = "
                  << other._length
                  << ", 释放已有资源. 重新申请资源 \
                  并复制源资源到目标资源" << endl;

            // 释放已有资源
            delete[] _data;

            _length = other._length;
            // 重新申请内存资源
            _data = new int[_length];
        }
    }

```

```

        // 复制源对象的内存资源到目标对象的内存资源
        copy(other._data, other._data + _length, _data);
    }
    // 返回重新申请内存资源后的对象
    return *this;
}

// 获取内存资源长度
size_t Length() const
{
    return _length;
}

private:
    size_t _length;        // 内存资源长度
    int* _data;            // 对象的内存资源
};

```

现在，可以使用 swap() 这个通用算法来交换两个 MemoryBlock 对象。

```

// 创建两个 MemoryBlock 对象
MemoryBlock a(3);
MemoryBlock b(6);
// 使用 swap() 算法交换两个对象
swap(a,b);

```

当运行这个程序时，它清楚地展示了两个对象交换过程中所进行的对象的创建、销毁以及内存资源的复制过程，输出如下：

```

创建对象，长度 = 3，申请资源
创建对象，长度 = 6，申请资源
复制构造函数 长度 = 3，申请资源并复制资源到目标资源
赋值操作符。长度 = 6，释放已有资源。重新申请资源
    并复制源资源到目标资源
赋值操作符。长度 = 3，释放已有资源。重新申请资源
    并复制源资源到目标资源
销毁对象，长度 = 3，释放资源
销毁对象，长度 = 3，释放资源
销毁对象，长度 = 6，释放资源

```

在这段代码中，虽然只是简单地交换了两个对象，但是通过输出结果可以发现，整个过程执行了多达 10 次的对象内存资源的申请与释放，同时进行了 3 次内存的复制操作。下面看看 swap() 算法的具体实现，从中了解为什么一个简单的对象交换会产生这么多操作。

```

// swap() 算法的具体实现
template <class T> void swap(T& a, T& b)
{
    T tmp(move(a));    // 对象 a 被复制到对象 tmp
    a = move(b);        // 对象 b 被复制到对象 a
}

```

```

        b = move(tmp);    // 对象 tmp 被复制到对象 b
    }

```

在 swap() 算法的实现中，使用了标准库中的 move() 函数来获取对象的右值引用，move() 只是简单地接受一个右值或者左值作为参数，然后直接返回相应对象的右值引用。这里可以清楚地看到，在进行对象交换的过程中，虽然利用 move() 函数获取对象的右值并利用它进行对象的构建和赋值，但是因为这些类型的变量无法充分利用右值，所以进行了很多多余的操作。同时，使用了一个中间变量，在整个过程中要对这个中间变量进行管理，还需要将资源在中间变量与目标变量之间进行反复复制，这无疑会影响程序的性能，特别是当这些对象比较大的时候，这种影响更加明显。

12.1.4 右值引用如何提高性能

右值引用又如何充分利用这些废弃的右值，以达到提高程序性能的目的呢？我们发现，通过赋值操作符“=”使用右值对变量进行赋值时，并没有直接使用这个即将废弃的右值将右值充分地利用起来，因此产生了多余的内存操作。通过为这个变量所属的类提供可以接受右值引用为参数的赋值操作符“=”和构造函数，实现对右值的废物再利用。

```

// 移动构造函数
// 接受右值引用为参数，将参数引用的对象移动作为要创建的目标对象
MemoryBlock(MemoryBlock&& other)
    : _data(NULL)
    , _length(0)
{
    cout << "右值引用构造函数，长度 = "
          << other._length << "，将资源指向已有资源" << endl;

    // 直接使用源对象的资源
    _data = other._data;
    _length = other._length;

    // 因为源对象是一个右值引用，
    // 所以将源对象的资源指针设置为空
    // 也就是将源对象清空，源对象不再可用
    other._data = NULL;
    other._length = 0;
}

// 移动赋值操作符“=”
// 接受右值引用为参数，直接将右值引用所指向的对象作为赋值完成后的对象
MemoryBlock& operator = (MemoryBlock&& other)
{
    if (this != &other)
    {
        cout << "右值引用赋值操作符，长度 = "
              << other._length << endl;
    }
}

```

```

        // 释放已有资源
        if ( NULL != _data )
        {
            cout<< "，释放已有资源，"<<endl;
            delete[] _data;
        }
        // 将资源指针直接指向源对象的资源
        _data = other._data;
        _length = other._length;

        // 因为源对象是一个右值，
        // 所以将源对象的资源指针设置为 NULL
        // 源对象不再可用
        other._data = NULL;
        other._length = 0;
    }

    return *this;
}

```

在这段代码中，为 MemoryBlock 类添加了一个移动构造函数和移动赋值操作符，这使得它具备了利用右值进行构造和赋值的能力。当用 move() 函数取得的右值进行对象的构造或者对其进行赋值的时候，因为 MemoryBlock 类提供了移动构造函数和移动赋值操作符，所以在整个过程中，它就可以直接移动这个源对象作为自己的对象本身，不会再产生新对象的创建与销毁，以及对象之间的内存资源的申请、释放和拷贝操作，减少了中间环节。现在的运行输出才是一个真正意义上的对象交换过程。其输出结果如下：

```

创建对象，长度 = 3，申请资源
创建对象，长度 = 6，申请资源
右值引用构造函数，长度 = 3，将资源指向已有资源
右值引用赋值操作符，长度 = 6
右值引用赋值操作符，长度 = 3
销毁对象，长度 = 0
销毁对象，长度 = 3，释放资源
销毁对象，长度 = 6，释放资源

```

从输出结果中可以看到，使用右值引用之后，申请资源和释放资源的次数从原来的 10 次减少为现在的 4 次，整个过程始终只有两份内存资源存在，避免了无谓的资源浪费。同时，现在对资源的重新指向代替了原来对资源的复制，这在很大程度上也提高了代码的性能。

通过提供可以接受右值引用为参数的构造函数和赋值操作符，就可以为程序带来非常明显的性能提升，这种好事当然不愿意错过。C++ 语言内建的数据类型已经支持右值引用。对于那些在 STL 算法中使用的用户自己创建的类，为它们提供可以接受右值引用为参数的构造函数和赋值操作符，同样可以充分利用右值，达到提高程序性能的目的。

12.2 智能指针 shared_ptr

要问为什么 C++ 语言具有如此高的性能，我想大多数程序高手都会说：C++ 语言可以通过指针直接操作内存，使得 C++ 语言具有了极高的性能，同时具备了极大的灵活性。

但是也应该看到，直接操作内存存在带来极高性能和极大灵活性的同时，因为内存操作的危险性，也给程序的健壮性带来了极大挑战，稍不留意，就可能导致内存访问错误。C++ 内存管理所带来的优势是如此诱人，使用时又是如此危险，如履薄冰，所以 C++ 程序员都是痛并快乐着。

12.2.1 C++ 的内存管理

在 C++ 语言中，可以使用 new 操作符和 malloc() 函数灵活自由地申请程序所需要的内存。例如，可以申请一段内存用来保存文件的路径名。

```
// 定义了_MAX_PATH 宏，表示文件路径名的最大长度
#include <stdlib.h>
// 引用定义了 malloc() 和 free() 函数的头文件
#include <stdio.h>
#include <malloc.h>
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 表示路径的指针
    char *string = NULL;

    // 为路径名这个字符串申请内存
    // 进行指针的类型转换
    string = (char *)malloc( _MAX_PATH );

    // 判断申请内存是否成功
    if( string == NULL )
        cout<<"无法获取内存"<<endl;
    else
    {
        cout<<"成功为路径名获取内存" <<endl;
        // 对获取的内存进行操作，
        // 例如按照某种规律构造路径名，用路径名保存文件等
        // ...
        // 释放申请的内存
        free( string );
        cout<<"释放内存" <<endl;
    }
    return 0;
}
```

这里，使用 `malloc()` 函数申请了一段长度为 `_MAX_PATH` 的内存，用于保存一个文件路径名字符串，并获得了指向这段内存的指针 `string`。通过 `string` 指针，就可以直接访问这段内存，并且按照某个规律灵活地构造路径名，或者使用这段内存保存的路径名来保存文件。直接操作字符串的内存，会比通过字符串对象的成员函数操作字符串带来更高的性能及更大的灵活性。

直接操作内存也带来了痛苦：当程序比较复杂时，这些指向内存片断、对内存进行管理的指针可能会在多个函数或者模块之间传递，这样就导致这些内存资源没有明确的所属对象，使得程序找不到合适的时机释放这些内存资源，而指向这些内存资源的指针就成为“野指针”。没有家长看管的孩子是“野孩子”，没有所属对象管理的指针就是“野指针”。

因为没有明确的所属关系，一些在多个模块中共享的资源指针很容易成为“野指针”，从而导致多种内存访问的问题。首先，因为共享某个内存块指针的多个模块并不拥有这个指针，它们只是使用这些指针，并不负责释放这些指针所指向的内存资源，这可能会导致程序的内存泄露。其次，因为这些指针被多个模块所共享，这可能会导致某个指针所指向的内存资源已经被释放，但是另外的模块又试图访问这个指针所指向的内存资源，最终导致内存访问错误。

C++语言对内存直接访问的支持使得程序员在享受内存直接访问所带来的高性能和高灵活性的同时，不得不小心地维护程序的健壮性。程序员都在盼望，如果指针可以更“聪明”一点，自己知道自己该什么时候释放资源就好了。作为程序员，只在需要的时候去申请内存资源，至于何时释放这些内存资源，就由这些“聪明”的指针自己去处理好了。程序员盼星星、盼月亮，终于盼来了完美解决这一问题的“聪明”的智能指针——`shared_ptr`。

12.2.2 用聪明的 `shared_ptr` 解决内存管理问题

为什么智能指针 `shared_ptr` 这么聪明，可以知道在合适的时机释放它所管理的内存资源？其实，`shared_ptr` 背后的管理机制并不复杂，它通过对所管理的内存资源进行引用计数来达到对这个资源的管理：当新增一个 `shared_ptr` 对该资源进行管理时，也就是新增一个指向此资源的 `shared_ptr` 时，就将该资源的引用计数加 1；反之，当减少一个 `shared_ptr` 对该资源进行管理时，就将该对象的引用计数减 1；如果该资源的引用计数为 0，说明没有任何指针对其进行管理，就自动调用 `delete` 释放其所占用的内存资源。例如，下面这段程序展示了引用计数的工作机制。

```
#include "stdafx.h"
// shared_ptr 定义在头文件<memory>中
#include <memory>
#include <iostream>
using namespace std;
```

```

int _tmain(int argc, _TCHAR* argv[])
{
    shared_ptr<int> pFirst( new int );
    // 这时，只有 pFirst 这个指针指向这块 int 类型的内存，
    // 所以这时的引用计数是 1
    cout<<"当前引用计数: "<<pFirst.use_count()<<endl;
    {
        // 创建另外一个 shared_ptr，并用 pFirst 对其进行赋值，
        // 让它们指向同一块内存资源
        shared_ptr<int> pCopy = pFirst;
        // 因为 pFirst 和 pCopy 都指向这一块内存资源，
        // 所以这一资源的引用计数增加为 2
        cout<<"当前引用计数: "<<pFirst.use_count()<<endl;
    }
    // 当超出 pCopy 的可见域，pCopy 结束其生命周期后，
    // 指向这一内存资源的只有 pFirst 指针，所以引用计数减为 1
    cout<<"当前引用计数: "<<pFirst.use_count()<<endl;

    // 当程序最终结束执行返回，pFirst 指针也结束其生命周期后，
    // 从此没有任何指针指向此内存资源，引用计数减为 0，内存资源自动得到释放
    return 0;
}

```

在这段程序中，首先，用 new 操作符申请了一块 int 类型的内存，用 pFirst 指针指向这块内存并对其进行管理。这时，指向这块内存的只有 pFirst 指针，所以引用计数为 1。然后，在一个局部作用域中，创建了另外一个 pCopy 指针，并用 pFirst 指针对其进行赋值，让它们都指向同一块内存。因为新增了 pCopy 指针指向这块内存，所以引用计数增加为 2。当超出 pCopy 指针的可见域时，pCopy 指针结束其生命周期，这时只有 pFirst 指针指向这块内存，所以引用计数减为 1。当程序最终结束执行返回时，pFirst 指针也结束其生命周期，从此没有任何指针指向此内存资源，引用计数减为 0，内存资源会自动得到释放。整个过程如图 12-2 所示。

几乎所有稍微复杂点的程序都需要某种形式的智能指针。如果没有智能指针的帮助，程序中对象的共享就会存在问题。因此，必须小心地进行内存资源的管理，编写专门的代码负责释放共享的内存。谁负责释放内存？什么时候释放内存？这都是让程序员头疼的问题。更重要的是，没有智能指针，就必须在内存之外增加生存周期的管理，这意味着在各拥有者之间存在更强的依赖关系，换言之，没有了重用性，增加了复杂性。

智能指针的出现很好地解决了这些内存管理问题，也增加了程序的可重用性，降低了程序的复杂度。这些智能指针可以不再为了控制两个或多个共享对象的生存周期而编写复杂的程序逻辑。当一个智能指针的引用计数降为 0，没有对象再需要这个共享的对象时，该对象就自动销毁。这是因为智能指针拥有与它所存指针有关的内存所有权，当它对对象进行销毁的时候，也会释放相应的内存资源。

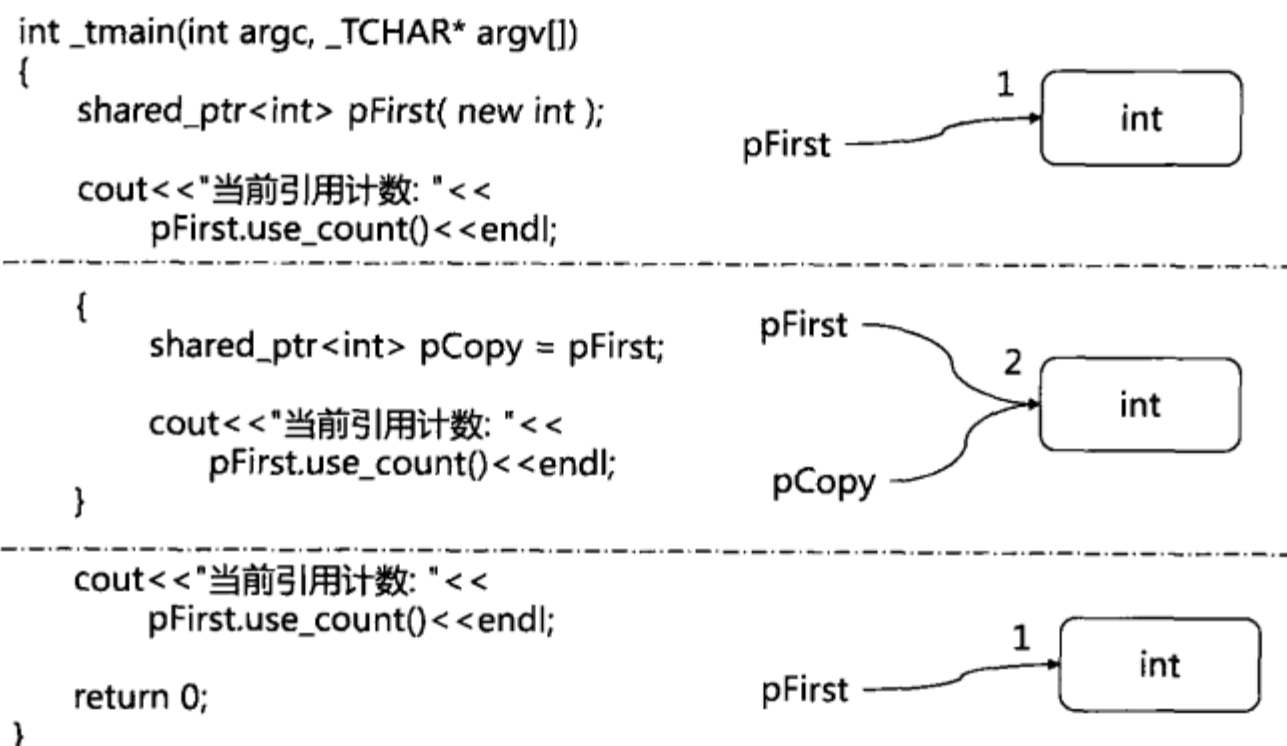


图 12-2 引用计数

12.2.3 智能指针的应用场景

既然智能指针可以给程序带来这么多好处，那么是不是在任何时候都应该使用智能指针呢？答案当然不是的。就像美食吃多了会坏肚子，美女看多了也会反胃，如果过分滥用智能指针，也会给程序带来负面的效应。只在合适的情况下使用智能指针，这样才能起到事半功倍的效果。

在所设计的对象中，如果有些特性使得它最好能够与智能指针一起使用，那么在这种情况下，智能指针的使用将使得这些对象的使用更加方便，两者相得益彰。例如，它的复制操作很昂贵，或者它所代表的有些资源必须共享多个实例。还有一种情形就是，共享的资源没有一个明确的拥有者。使用智能指针可以在需要访问共享资源的对象之间共享资源的所有权。智能指针还可以把对象指针包装后存入标准库的容器中而不会有任何内存泄漏的风险，特别是在面对异常或要从容器中删除元素的时候。如果把指针放入容器，就可以获得多态的好处，可以提高性能。如果复制的代价很高，那么性能的提高将非常明显。另外，还可以通过把相同的对象放入多个辅助容器来进行特定的查找。

总结起来，当出现以下情况时应该优先考虑使用 `shared_ptr`：

- 有多个使用者共同使用同一个对象，而没有一个明显的拥有者；
- 一个对象的复制操作很昂贵；
- 要把指针存入标准库容器；
- 要传送对象到库或从库获取对象，而这些对象没有明确的所有权；

- 当管理需要特殊清除方式的资源时，这时可以通过定制 `shared_ptr` 的删除器来实现。

12.2.4 `shared_ptr` 的使用

`shared_ptr` 实际上是一个类模板，就像需要指定指针的数据类型一样，也需要使用某个特定的数据类型来实例化这个类模板，以形成特定的可以指向这个具体数据类型的 `shared_ptr`。例如：

```
// 使用 int 实例化 shared_ptr, spInt 可以指向一个 int 对象
shared_ptr<int> spInt;
// 使用 Employee 实例化 shared_ptr, spEmployee 可以指向一个 Employee 对象
shared_ptr<Employee> spEmployee;
```

实例化特定的 `shared_ptr` 模板类之后，可以开始用这些模板类构造具体的 `shared_ptr` 对象。`shared_ptr` 提供了多个构造函数，可以灵活地完成 `shared_ptr` 的构造。比如，可以从一个裸指针、一个 `shared_ptr`，或者一个 `weak_ptr` 构造相应的 `shared_ptr` 对象，甚至可以为 `shared_ptr` 提供一个删除器，以完成删除动作的自定义。

1. `template<class Other> explicit shared_ptr(Other * ptr);`

这个构造函数获得给定指针 `ptr` 的所有权。参数 `ptr` 必须是指向 `Other` 的有效指针。构造后引用计数设为 1，构造完成后将获得一个跟 `ptr` 指向相同对象的 `shared_ptr`。例如：

```
Employee* pEmployee = new Employee;
// 使用裸指针 pEmployee 构造 shared_ptr
shared_ptr<Employee> spEmployee1( pEmployee );
// 直接使用 new 操作符返回的指针构造 shared_ptr
shared_ptr<Employee> spEmployee2( new Employee );
```

2. `shared_ptr(const shared_ptr& sp);`

这个构造函数可以方便使用一个 `shared_ptr` 构造另外一个 `shared_ptr`，原有 `shared_ptr` 中保存的资源将被新构造的 `shared_ptr` 所共享，引用计数加 1。例如：

```
// 使用 spEmployee1 构造 spEmployee3，两者将指向同一个对象，引用计数增加 1
shared_ptr<Employee> spEmployee3( spEmployee1 );
```

`shared_ptr` 对象创建成功后，它就可以像普通指针一样使用了。唯一不同的是，因为它是智能指针，它不需要也不能够被显式地删除。只需要使用 `shared_ptr` 就可，至于删除的事情，就让它自己去处理好了。例如：

```
// 定义一个 shared_ptr
shared_ptr<Employee> spEmployee1( pEmployee );
// 使用 “->” 操作符，可以将 shared_ptr 当成普通指针一样使用，
// 访问它所指向的对象
int nAge = spEmployee1->GetAge();
// PrintObj() 函数接受一个 Employee 对象作为参数，
```

```
// 使用 “*” 操作符，可以获得 shared_ptr 所指向的 Employee 对象
// PrintObj() 函数需要的参数是一个 Employee 对象
PrintObj(*spEmployee1);
```

在这里，使用了 shared_ptr 最常用的两个操作符。“*” 运算符会返回对 shared_ptr 已存指针所指向对象的一个引用，如果再配合 “&” 运算符，就可以得到这个 shared_ptr 所管理的指针。例如：

```
// 一个需要指针作为参数的函数
void PrintPtr (Employee* pEmp )
{
    // 函数体
}
// 通过 “&*” 运算符获得 shared_ptr 所管理的指针，
// 并作为参数传递给函数
PrintPtr( &*spEmployee );
```

如果在 shared_ptr 之后使用 “->” 运算符，它将返回智能指针所保存的指针。这个运算与 “*” 一起使得智能指针看起来像普通指针一样。

既然是智能指针，除了使用上跟普通指针相似之外，当然也有不同于普通指针的地方，除了提供 “*” 和 “->” 两个常用的运算符之外，它还提供了很多成员函数以方便我们使用。

3. “=” 赋值操作符

“=” 赋值操作符会让 shared_ptr 共享新的资源，并停止对原有资源的共享。简单地讲，“=” 赋值操作符将结束一个智能指针原有的指向，指向新的资源。例如：

```
shared_ptr<Employee> spEmployee1( new Employee("Jiawei") );
shared_ptr<Employee> spEmployee2( new Employee("ChenLiangqiao") );
// 构造一个空的 shared_ptr，不指向任何对象
shared_ptr<Employee> spEmployee0;
// 将 spEmployee0 指向 spEmployee1 所指向的对象
// 这时 spEmployee1 的引用计数是 2
spEmployee0 = spEmployee1;
cout<<"员工的名字是"<<spEmployee0->GetName()<<endl;
// 结束 spEmployee0 对 spEmployee1 的指向，
// 将 spEmployee0 指向 spEmployee2 所指向的对象
// 这时 spEmployee1 的引用计数减少为 1，spEmployee2 的引用计数增加为 2
spEmployee0 = spEmployee2;
cout<<"员工的名字是"<<spEmployee0->GetName()<<endl;
```

4. reset() 函数

shared_ptr 的 reset() 函数可以用于对一个 shared_ptr 进行重新设置，停止对 shared_ptr 所保存指针所有权的共享，也就是断开 shared_ptr 与原有资源的联系，共享资源的引用计数减 1。也可以通过 reset() 函数的参数让 shared_ptr 指向新的共享资源。例如：

```

shared_ptr<Employee> spEmployee1( new Employee("Jiawei") );
// 断开 spEmployee1 与 Employee 对象的联系, 引用计数减 1
spEmployee1.reset();
// 将 spEmployee1 重新指向另外一个 Employee 对象
spEmployee1.reset( new Employee("ChenLiangqiao") );

```

5. get()函数

虽然可以像普通指针一样使用 `shared_ptr`, 但是有时能得到 `shared_ptr` 所保存的普通指针还是非常有用的。例如, 当 `shared_ptr` 所保存的指针可能为空时, 如果使用 “*” 和 “->” 两个运算符来获取这个指针所指向的对象, 则会导致未定义行为。在这种情况下, 获取 `shared_ptr` 所指向对象的最好办法就是通过 `get()` 函数获取 `shared_ptr` 所保存的指针, 进而判断指针的有效性, 从而安全地获得 `shared_ptr` 所指向的对象。注意, 也可以使用隐式 `bool` 类型转换来测试 `shared_ptr` 是否包含有效指针。例如:

```

shared_ptr<Employee> spEmployee1( new Employee("Jiawei") );
// 通过 get() 函数获得 shared_ptr 所保存的指针
Employee* pEmp = spEmployee1.get();
// 通过隐式类型转换, 将 shared_ptr 转换为 bool 类型, 判断 shared_ptr 的有效性
// 等效于 if( true == spEmployee1.get() )
if( true == (bool)spEmployee1 )
    PrintObj(*spEmployee1); // 安全地进行 “*” 操作

```

有时候, 也使用 `get()` 函数来获得 `shared_ptr` 所保存的指针, 以传递给那些只能接受普通指针的函数。除了 `get()` 函数之外, 还可以在 `shared_ptr` 之前使用 “&*” 运算符来获得这个 `shared_ptr` 所管理的对象指针。

6. use_count()函数

`use_count()` 函数返回 `shared_ptr` 的引用计数。它在调试的时候特别有用, 因为它可以在程序执行的关键点获得引用计数的快照, 还可以为调试提供非常有用的信息。

7. shared_ptr::operator boolean-type

这是个将智能指针转换到 `boolean-type` 类型的隐式类型转换函数, 它可以在 `boolean` 上下文中测试智能指针, 从而判断智能指针的有效性。如果 `shared_ptr` 保存一个有效的指针, 则返回值为 `true`; 否则为 `false`。这里需要注意的是, 转换函数返回的类型是不确定的。很多时候需要显式地将结果转换为 `bool` 类型。例如:

```

// 判断 spEmployee1 的有效性
if( true == (bool)spEmployee1 )
    PrintObj(*spEmployee1);

```

8. swap()函数

`swap()` 函数可以很方便地交换两个 `shared_ptr`。`swap()` 函数不仅交换 `shared_ptr` 所保存的指针, 也会改变各自的引用计数。例如:

```

shared_ptr<Employee> spEmployee1( new Employee("Jiawei") );
shared_ptr<Employee> spEmployee2( new Employee("ChenLiangqiao") );
// 交换 spEmployee1 和 spEmployee2 所指向的对象, 同时改变它们的引用计数
// 交换之后, spEmployee1 指向 Employee("ChenLiangqiao"),
// 而 spEmployee2 则指向 Employee("Jiawei")
spEmployee1.swap( spEmployee2 );

```

shared_ptr 这种使用引用计数的资源管理机制及它所提供的各种功能函数可以很好地解决 C++ 语言的内存管理问题, 特别是解决那些被一个或者多个客户共享的资源在什么时候、如何正确释放的问题。下面的例子演示了如何使用 shared_ptr 管理多个客户所共享的资源。

```

// 引入需要使用的头文件
#include <memory>
#include <iostream>
#include <string>
using namespace std;

// 一个简单的员工类, 也是要共享的对象类
class Employee
{
public:
    Employee(string strName )
    {
        m_strName = strName;
    }
    string GetName() { return m_strName; };
private:
    string m_strName;
};

// 共享员工对象的类
// 负责使用英文格式打印员工信息
class PrintEng
{
public:
    // 将 shared_ptr 传递给构造函数, 在多个客户之间共享对象
    PrintEng(shared_ptr<Employee> sp ) : m_spEmp(sp) {};
    void doPrint()
    {
        // 使用 shared_ptr 访问共享的 Employee 对象
        // 以英文格式输出
        if( true == (bool)m_spEmp)
            cout<<"Name of Employee:"<<m_spEmp->GetName()<<endl;
    }
private:
    // 用于保存共享对象的 shared_ptr
    shared_ptr<Employee> m_spEmp;
};

// 共享员工对象的类
// 负责使用中文格式打印员工信息
class PrintChs

```

```

{
public:
    // 将 shared_ptr 传递给构造函数，在多个客户之间共享对象
    PrintChs(shared_ptr<Employee> sp) : m_spEmp(sp) {};
    void doPrint()
    {
        // 使用 shared_ptr 访问共享的 Employee 对象
        // 以中文格式输出
        if( true == (bool)m_spEmp)
            cout<<"员工的姓名是："<<m_spEmp->GetName()<<endl;
    }
private:
    // 用于保存共享对象的 shared_ptr
    shared_ptr<Employee> m_spEmp;
};

int _tmain(int argc, _TCHAR* argv[])
{
    // 创建一个新的 shared_ptr，用它来管理 Employee 对象
    shared_ptr<Employee> spEmp(new Employee("Jiawei"));
    // 将 spEmp 所管理的 Employee 对象共享给 pEng 对象
    PrintEng pEng(spEmp);
    pEng.doPrint();
    // 将 spEmp 所管理的 Employee 对象共享给 pChs 对象
    PrintChs pChs(spEmp);
    pChs.doPrint();

    // 当 spEmp、pEng 和 pChs 都退出作用域，结束其生命周期时，
    // 最后一个 shared_ptr 将负责释放其管理的 Employee 对象
    return 0;
}

```

在这段代码中，首先，创建了一个简单的 Employee 类作为被管理的对象。然后，创建了两个打印类，分别负责以英文和中文格式打印员工信息。因为打印时需要员工的信息，所以它们都以 shared_ptr 的形式来共享外部提供的 Employee 对象。在主函数中，首先创建一个 shared_ptr 智能指针 spEmp，并用它来管理一个 Employee 对象；然后用 spEmp 对象来创建两个打印类对象 pEng 和 pChs，以此来实现现在 pEng 和 pChs 中共享 Employee 对象。到目前为止，利用 shared_ptr 实现了 Employee 对象的共享：spEmp、pEng 和 pChs 对象的 m_spEmp 都指向同一个 Employee 对象，多个客户共享同一个对象。在这里，虽然可以使用普通指针来实现对象的共享，但是在删除对象的时候，这个被共享的 Employee 对象并没有明确的拥有者，当对象使用完成之后，谁来负责删除这个对象将成为一个“大”问题。

当使用 shared_ptr 对这个对象进行管理时，不仅可以像使用普通指针一样使用 shared_ptr 来访问这个被管理的对象。更重要的是，如果所有的 shared_ptr 离开其作用域、结束其生命周期，那么 shared_ptr 的引用将减少为 0，而最后一个 shared_ptr 将负责共享的 Employee 对象的删除，从而一劳永逸地解决了共享对象的删除问题。

虽然智能指针 `shared_ptr` 通过引用计数的方式完美地解决了 C++ 语言中的内存管理问题,但是,同样是因为引用计数,却又给 `shared_ptr` 带来两个新问题。

1. 体积问题

因为 `shared_ptr` 需要进行引用计数,所以它需要额外的内存空间来保存当前内存资源的引用数,这使得一个 `shared_ptr` 指针将占用 40 个字节的内存空间,是一个普通指针所占用内存空间的整整 10 倍。如果要在程序中保存和处理大量的 `shared_ptr`, `shared_ptr` 的体积问题将是一个“大”问题。

2. 性能问题

一方面,因为引用计数, `shared_ptr` 需要在运行时对引用计数进行管理和维护,这样就必然需要消耗额外的计算资源,影响程序的性能。另一方面, `shared_ptr` 使用了大量的虚函数,虚函数的使用也带来了一定的性能损失。

这真是成也“引用计数”,败也“引用计数”。不过不用担心, C++ 为我们提供了一个瘦身版的智能指针——`unique_ptr`。使用 `unique_ptr`, 可以很好地解决 `shared_ptr` 因为引用计数所遇到的这些问题。跟 `shared_ptr` 一样, `unique_ptr` 也可以对它所关联的内存资源进行管理,当 `unique_ptr` 销毁时,同样会自动释放它所管理的内存资源。但与 `shared_ptr` 不同的是, `unique_ptr` 没有引用计数,也就是说,某一个内存资源,只允许有一个 `unique_ptr` 与之关联,对其进行管理。因为没有引用计数,这就使得 `unique_ptr` 避免了 `shared_ptr` 的体积问题和性能问题,成为 `shared_ptr` 在某些情况(需要使用大量智能指针而又无须对智能指针进行复制)下的一个很好的替代。在使用上, `unique_ptr` 跟 `shared_ptr` 基本相同,两者唯一的差别就是, `unique_ptr` 不可以进行复制,换句话说,不能有两个 `unique_ptr` 关联同一内存资源。例如:

```
// 定义一个 unique_ptr, 并将其与一个 Employee 对象关联
unique_ptr<Employee> upEmployee( new Employee );
// 使用 “->” 运算符, 直接访问 unique_ptr 所指向对象的成员函数
int nAge = upEmployee->GetAge();
// 使用 “*” 运算符, 获得 unique_ptr 所指向的对象
PrintObj(*upEmployee);
```

12.2.5 shared_ptr 与标准库容器

在程序中,我们常常利用容器对多个相似的对象进行管理。如果把这些对象直接存入容器中,那么会给程序带来麻烦。首先,如果以值的方式将对象保存到容器中,则意味着使用者获得的是容器中元素的拷贝而不是元素本身;对于那些复制是一种昂贵的操作的对象来说,则可能会有性能上的损失。有些容器,特别是 `vector` 容器,当加入对象到容器中时可能会复制所有元素,这更加重了性能的问题。其次,传值意味着没有多态的行为。如果需要在容器中存放多态的对象且不想切割它们,则必须使用指针。但是,如果使用裸指针将对象保存到

容器中，维护元素的完整性则会非常复杂。从容器中删除元素时，必须知道容器的使用者是否还在使用将要被删除的元素，否则可能会错误地删除某个正在使用中的元素，引起严重的内存访问错误。现在，这些问题都可以用 `shared_ptr` 轻松解决。

使用 `shared_ptr` 代替普通指针，以智能指针的形式将对象保存到容器中，这样就可以解决容器使用过程中因为复制元素而引起的性能问题，因为容器中保存的元素都是智能指针，所以复制元素时并不会存在性能问题。同时，智能指针在使用完成后，它所管理的资源将被自动安全地释放，无须对其完整性进行额外的维护。下面的例子演示了如何在容器中使用 `shared_ptr`。

```
int _tmain(int argc, _TCHAR* argv[])
{
    // 为了书写简便，使用 typedef 重新定义可以保存 shared_ptr<Employee> 智能指针的
    // vector 容器为新的容器类型 container
    typedef vector<shared_ptr<Employee>> container;

    // 创建 shared_ptr 对象
    shared_ptr<Employee> spEmp(new Employee("Jiawei"));
    // 创建容器，容器类型为 container，
    // 实质上是 vector<shared_ptr<Employee>>
    container vecEmp;
    // 将 shared_ptr 保存到容器中
    vecEmp.push_back( spEmp );
    vecEmp.push_back( shared_ptr<Employee>( new
        Employee("ChenLiangqiao") ) );
    // 遍历容器中的元素，访问其中的智能指针
    for( container::iterator it = vecEmp.begin();
        it != vecEmp.end(); ++it )
    {
        // 通过迭代器 it 获得容器中保存的智能指针
        PrintEng pEng(*it);
        pEng.doPrint();
    }

    return 0;
}
```

从以上代码中可以看到，智能指针 `shared_ptr` 的使用跟普通指针相似。同样以 `shared_ptr` 作为容器元素的类型，使用同样的方法将 `shared_ptr` 添加到容器中；同样使用迭代器来获得容器中保存的 `shared_ptr` 元素。`shared_ptr` 的使用跟普通指针的使用一致，但是会带来很多额外的效益。如果用裸指针作为元素将对象保存到容器中，那么在对象使用完成之后，需要手工删除这些对象。在这个例子中，因为使用 `shared_ptr` 代替了普通指针，所以这些对象的删除就变成了完全自动的，因为在 `vector` 的生存期中，每个 `shared_ptr` 的引用计数都保持为 1；当 `vector` 被销毁、所有 `shared_ptr` 的引用计数都变为 0 时，所有对象都会自动删除。使用 `shared_ptr` 代替普通指针作为容器的元素，使用方法跟普通指针完全一致而且能带来这么多好处，何乐而不为呢？

12.2.6 对 shared_ptr 进行自定义

有时候，使用 shared_ptr 管理某些特别的类型时，这些特别的类型对象的释放不是简单地使用 delete 关键字释放就可以完成的，它们的清理还需要一些额外的工作。在这种情况下，需要对 shared_ptr 进行自定义以完成特殊的清理工作。

shared_ptr 所管理的资源的清理工作都是由删除器 (deleter) 来完成的。shared_ptr 提供了一些特殊的构造函数，这些构造函数可以指定 shared_ptr 的删除器，从而对 shared_ptr 的清理工作进行自定义。

```
template<class Other, class D> shared_ptr(Other * ptr, D dtor);
```

这个构造函数带有两个参数。第一个参数是 shared_ptr 将要获得所有权的某个资源；第二个参数则是 shared_ptr 被销毁时负责释放资源的一个对象，它称为删除器。删除器会在 shared_ptr 的引用计数变为 0 时被调用，被保存的资源将以 dtor(ptr) 的形式传给该删除器，用来处理共享资源的自定义释放。这对于管理那些不是用 new 分配也不是用 delete 释放的资源时非常有用。例如，清理一个文件对象时，往往不是使用 delete 操作符释放对象，而是使用相应的成员函数关闭文件。下面的例子演示了如何通过自定义 shared_ptr 来完成特殊的清理工作。

```
// 引入需要的头文件
#include "stdafx.h"
// 文件操作需要的头文件
#include <stdio.h>
// 获取系统时间需要的头文件
#include <time.h>
#include <memory>
#include <iostream>
using namespace std;

// 使用一个函数对象来表示自定义的删除器
class FileCloser
{
public:
    // 重载 "()" 操作符，在其中定义指针的特殊清理动作
    void operator()(FILE* pFile)
    {
        // 判断指针是否为 NULL
        if( NULL != pFile )
        {
            // 写入日志文件结束的标志
            fprintf( pFile, "日志文件结束\n" );
            // 输出提示日志文件被关闭
            cout<<"关闭日志文件"<<endl;
            // 关闭日志文件，完成文件指针的清理工作
            fclose(pFile);
        }
    }
}
```

```

};

// 使用日志文件这一文件指针的多个客户
// 它们通过 shared_ptr 共享同一个日志文件，分别向同一个日志文件打印系统时间和系统日期
// 向日志文件打印系统时间
void PrintTime( shared_ptr<FILE> spLog )
{
    char tmpbuf[128];
    // 获取系统时间
    strtime_s( tmpbuf, 128 );
    // 将系统时间输出到共享的日志文件
    fprintf( spLog.get(), "当前系统时间: \t\t%s\n", tmpbuf );
}
// 向日志文件打印系统日期
void PrintDate( shared_ptr<FILE> spLog )
{
    char tmpbuf[128];
    // 获取系统日期
    _strdate_s( tmpbuf, 128 );
    // 将系统日期输出到共享的日志文件
    fprintf( &*spLog, "当前系统日期: \t\t%s\n", tmpbuf );
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 创建并打开日志文件
    FILE* fLog = NULL;
    int err = fopen_s(&fLog, "log.txt", "w");

    // 成功打开日志文件
    if ( 0 == err )
    {
        cout<<"打开日志文件并写入日志"<<endl;
        // 使用 shared_ptr 管理文件指针
        shared_ptr<FILE> spLog(fLog, FileCloser() );

        // 直接使用 shared_ptr 访问文件指针输出日志
        fprintf( spLog.get(), "日志文件开始\n" );

        // 通过 shared_ptr 实现文件指针在多个客户之间的共享
        PrintTime( spLog );
        PrintDate( spLog );
    }

    return 0;
}

```

在这段代码中，首先定义了一个函数对象 FileCloser，它就是自定义的删除器。通过重载它的“()”操作符，在其中调用 fclose()函数关闭文件，完成资源的特殊清理工作。当进行清理工作时，因为 shared_ptr 会将它所管理的指针作为参数传递给删除器对象，所以操作符“()”

的参数类型就是它要清理的指针，在这里，也就是 `shared_ptr` 所管理的 `FILE*` 类型。当创建智能指针 `shared_ptr` 时，通过指定自定义的删除器，可以让 `shared_ptr` 在清理资源的时候进行特殊的操作，从而完成 `shared_ptr` 的自定义。在主函数中，通过 `shared_ptr` 的构造函数不仅可以指定 `shared_ptr` 所管理的指针，而且可以指定其删除器为 `FileCloser()`，这样 `shared_ptr` 在释放资源的时候，就会以它所管理的指针作为参数调用这个函数对象，在其中完成特殊的清理工作，如图 12-3 所示。

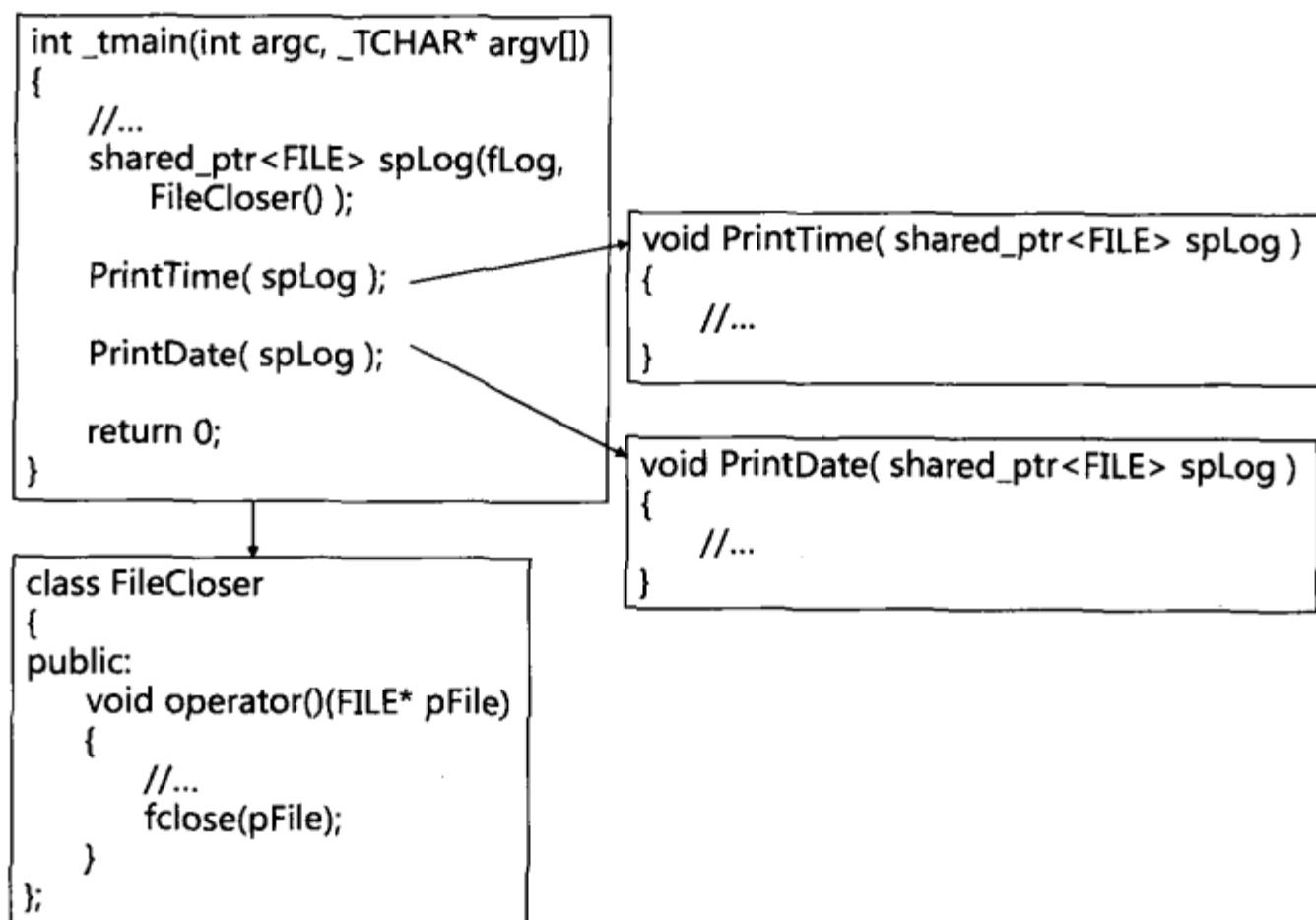


图 12-3 `shared_ptr` 的使用与销毁过程

这里值得注意的是，当通过 `shared_ptr` 访问资源时，需要对 `shared_ptr` 使用“&*”用法，或者使用它的 `get()` 函数来获得它所管理的指针。另外，如果在释放资源时只需要调用一个单参数函数，就不必大费周折地去创建一个客户化删除器类型，只需要传递给 `shared_ptr` 的构造函数这个删除函数的指针就可。上面的例子可以重写如下：

```
// 调用 fclose() 函数完成清理工作
shared_ptr<FILE> spLog(fLog, &fclose);
```

通过使用定制的删除器来完成 `shared_ptr` 的自定义，几乎所有的资源类型都可以使用 `shared_ptr` 进行的管理。这使得 `shared_ptr` 成为对共享资源进行管理的通用类，而不仅仅是处理动态分配的对象。与普通的裸指针相比，`shared_ptr` 会付出一点点额外的空间代价，但是跟它所带来的内存管理方面的好处比起来，这一点点代价是值得的。

12.3 用 PPL 进行多线程开发

“我的程序怎么这么慢啊？”

“为什么 CPU 的使用率只有 50%呢？”

没错，你使用的是一个单线程的应用程序。单线程的应用程序只能运行在 CPU 的一个运算核心，而对于现在普遍的多核 CPU 来说，这是很浪费的。就像 CPU 团队中有好几个人可以干活，却只让一个人拼命干活，其他几个人闲着一样，这样当然会导致应用程序的性能低下。俗话说，人多力量大。只有把大家都调动起来，才能又快又好地完成任务，而这个可以把大家都调动起来的法宝就是 PPL（parallel patterns library）。

12.3.1 多核给程序设计带来的挑战

小时候，老师总教育我们上课要专心，“一心不可二用”。可是这个道理在计算机的世界却行不通，因为 CPU 这个不听话的孩子，偏偏在单核 CPU 之外发展出多核 CPU，总想着一“芯”多用。随着多核 CPU 的推出，一“芯”多用已成为越来越普遍的事情。从单核到双核，从双核到四核再到八核等，开始进入一个一芯多核的时代，如图 12-4 所示。

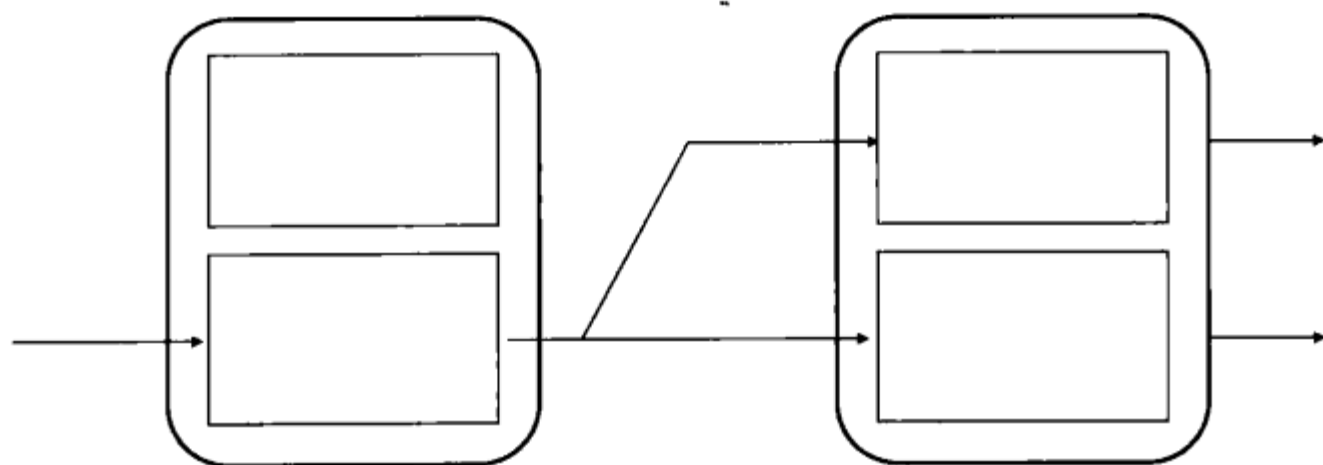


图 12-4 程序在多核 CPU 的串行计算和并行计算

在以往的计算机发展历史中，硬件技术的发展，特别是 CPU 频率的不断提升，总是免费提升软件的性能。从 386 到 586，从赛扬到奔腾，每次 CPU 频率的提升，都给软件性能带来大幅提升，而软件无需做任何变动。如果有用户抱怨你的软件性能不佳，则无须着急，只需要坐等 Intel 或者 AMD 推出更高频率的 CPU 就可以了。程序员把这形象地称为“免费的午餐”。

但是，当单核 CPU 的频率发展到 3G 时，再往上发展就遇到了技术瓶颈。因为单核 CPU 的发展已经达到一个极限，所以硬件厂商不得不转向多核 CPU 的发展，在一个芯片中加入两个甚至多个运算核心，这样通过核心的增加来提高芯片总的频率。虽然在硬件厂商看来这仿

佛是一个创举，但是却给软件厂商带来了无尽的烦恼。程序员发现，进入多核时代，这种“免费的午餐”再也没有了。其中最主要的原因就是当前的应用程序几乎都是针对一个运算核心而设计的，当硬件通过增加运算核心来提高性能时，由于受到其架构的影响，软件并不能充分地利用多个运算核心所带来的性能提升，甚至有时性能还有所下降。

软件是运行在硬件之上的。当硬件发展时，软件的架构和开发方法也要做相应的变化，只有这样才能跟得上时代的步伐。多核时代的到来，软件世界的并行计算开始兴盛起来。如何充分地利用 CPU 的多个核心？如何发挥多核的威力，带来软件性能上的提升？如何改造现有的程序，让它更好地适应多核时代？如何简便地实现并行计算，使自己开发的应用程序充分利用硬件升级所带来的性能提升？这些问题，无不困扰多核时代的程序员。

12.3.2 PPL 带来免费的午餐

人在饥饿的时候智慧是无穷的，程序员也不例外。为了继续吃上这免费的午餐，程序员想出了一个好办法：利用多线程将应用程序并行化，充分利用 CPU 的多个运算核心，这样免费的午餐又回来了。

多线程是这样一种机制，它允许在程序中并发执行多个指令流，每个指令流称为一个线程，彼此间互相独立。线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度。两者的区别在于线程没有独立的存储空间，而是和所属进程中的其他线程共享一个存储空间。

多个线程的执行是并发的，也就是逻辑上的“同时”，在多核 CPU 上，这些线程的执行也可能是物理上的“同时”，因为多个线程可以同时运行在 CPU 的多个运算核心上。正是因为多线程应用程序在逻辑上和物理上的这种并发性，它把程序要完成的任务分解为多个独立的事务，每个事物独立完成，这就像本来一个人干的活分给了多个人一起干，极大地提升了应用程序的性能：

- 充分利用多核 CPU 的计算能力，不浪费硬件资源；
- 多线程技术可以及时响应程序，因为当其他工作继续时用户界面可以保持激活状态；
- 当前不繁忙的事务可以把 CPU 时间让给其他事务，这样可以保证事务的及时执行；
- 花费大量处理时间的事务可以周期性地把时间让给其他的事务；
- 事务可以在任何需要的时候停止；
- 可以通过把单独事务的优先级调高或调低来优化性能。

毫无疑问，利用多线程将应用程序并行化，可以充分利用 CPU 的硬件资源，极大地提升

应用程序的性能。但是，也并不是任何程序都需要并行化，即使并行化，也并不是线程越多越好。如果使用不当，多线程也可能会成为一个负担或者需要付出不小的代价。如果一个程序有很多的线程，那么其他程序的线程必然只能占用更少的 CPU 资源，而且大量的 CPU 资源会用于线程的调度，操作系统也需要足够的内存空间来维护每个线程的上下文信息。由此可见，大量的线程会降低系统的运行效率。

那么，该在什么时候使用多线程呢？

- 程序中需要大量的运算，比如图像处理、三维建模等；
- 时间密集或处理密集的事务妨碍用户界面的响应；
- 单独的事务必须等待外部速度较慢的资源，例如打印任务需要等待打印机响应。

如果使用多线程，则程序的多线程必须设计得很好，否则所带来的好处将远小于坏处。因此，使用多线程必须小心地处理这些线程的创建、调度和释放等工作。

虽然多线程能够极大地提升应用程序的性能，也能够改善应用程序的用户体验，但是多线程应用程序的开发相对比较烦琐，对开发者也提出了更高的要求。有没有一种更加简单的方法可以进行多线程应用程序的开发？

有！没错，每当有一件烦琐的事情出现时，就有一个可以让事情变得更加简单的解决办法出现，而 PPL 就是多线程开发的解决办法。

PPL 是微软在 Visual C++ 2010 中提供的一个简化多线程应用程序开发的编程模型。PPL 建立在并发运行时（concurrency runtime）的调度和资源管理组件之上。它在代码与底层线程机制之间插入了一层抽象层，提供支持并发的泛化、类型安全的算法和并行容器。

PPL 支持如下特性。

- 并行算法：并行作用于一组数据的泛化算法。
- 并行任务：一个可并行执行几个工作单位（任务）的机制。
- 并行容器和对象：可安全地并发存取其元素的泛化容器。

PPL 提供的编程模型类似于标准模板库。并行算法可以自动地对一组可以并行处理的数据进行切分，并创建合适的进程对切分后的数据进行并行处理。并行容器支持数据的并行访问。并行任务则可以通过任务的机制简单地将一个要完成的工作并行化，无须我们去进行具体工作线程的创建和管理等，极大地减少了多线程开发的工作量。于是 PPL 自豪地对程序员说：免费的午餐又回来了。

12.3.3 PPL 中的并行算法

通常，我们所要处理的数据是相互独立、可以并行处理的。针对这类数据，PPL 中的并行算法可以非常简单地将这些数据并行化，从而完成算法的并行化。在这个过程中，无须去处理数据的划分、线程的创建和管理等一系列烦琐的工作，这一切都将由 PPL 代劳。PPL 会自动根据 CPU 的资源进行数据的划分，并创建合适的线程对数据进行处理，只需要坐享其成就可以了。PPL 提供了以下三种并行算法。

1. parallel_for 算法

parallel_for() 算法提供了对常用的 for 循环语句的并行化支持。比如，在程序中有这样一个 for 循环：

```
// 标准 for 循环
for( int i = 0; i < 100; ++i )
{
    // 循环体语句
}

使用 parallel_for() 算法，可以非常简单地将这个标准 for 循环并行化。
// 并行化的 for 循环
parallel_for( 0, 100, 1,
    [&](int i)      // parallel_for() 算法的 Lambda 表达式
    {
        // 循环体语句
    });
```

这里可以看到，parallel_for() 算法接受 4 个参数，分别是 for 循环的起始值、终止值、步长及表示 for 循环体的 Lambda 表达式。通过使用 parallel_for() 算法，现在这个 for 循环将被并行地执行，而具体的线程数，PPL 将根据 CPU 的资源自动地做出决定。

2. parallel_for_each 算法

除了标准的 for 循环之外，通常需要处理的数据就是 STL 容器中的数据。在 STL 中，通常使用 for_each() 算法访问容器中的各个元素。但是这些访问都是串行的，对于大多数可以并行处理的数据，这无疑会浪费 CPU 的资源。而 parallel_for_each() 算法可以说是并行版本的 for_each() 算法，它可以并行处理一个容器中的数据，以达到充分利用 CPU 资源的效果。例如，可以这样来并行处理 vector 容器中的数据：

```
// 并行处理容器中的数据
vector<int> vec;
parallel_for_each(vec.begin(), vec.end(),
    [&](int& i)      // parallel_for_each() 算法的 Lambda 表达式
    {
        // 对容器数据进行处理
    });
```

跟 `for_each()` 算法相似, `parallel_for_each()` 算法可以接受 3 个参数, 分别是容器的起点、容器的终点及对容器元素进行处理的 Lambda 表达式。通过 `parallel_for_each()` 算法的应用, 可以轻松地实现对容器元素的并行访问。

3. `parallel_invoke()` 算法

在多线程开发中, 如果想创建一个工作者线程来完成一定的任务, 不仅要编写线程函数, 还要进行线程的创建和管理。现在, 有了 `parallel_invoke()` 算法, 一切都变得那么简单。使用 `parallel_invoke()` 算法, 只需要编写线程的线程函数, 然后委托给它去执行就可以了。`parallel_invoke()` 算法会完成线程的创建管理等烦琐工作, 我们可以把主要精力集中在更重要的线程函数的编写上。例如:

```
// 并行执行任务
parallel_invoke(
    [=]{for (int i=0; i<100; ++i) // 任务 1: 输出 0 到 100 的整数
        {cout << i << endl; }},
    [=]{for (int i=0; i<10; ++i) // 任务 2: 输出 10 个空格
        {cout << "    " << i << endl; }}
);
```

`parallel_invoke()` 算法可以接受 2~10 个函数对象作为参数, 这些函数就是各个线程的线程函数, `parallel_invoke()` 算法会自动为每个函数创建相应的线程来完成函数的执行。这样, 我们就可以轻松地完成多个并行任务, 而不用去管什么线程的创建和管理等琐事了。

通常我们编写的计算 PI 值的程序, 虽然能够达到很高的精度, 但是因为所采用的是一个串行的算法, 所以效率不是很高, 往往需要比较长的时间才可以得到比较精确的值。现在, 可以利用 PPL 中的并行算法来改写计算 PI 值的程序, 让这个程序变得又好又快。

```
#include "stdafx.h"
// 引入 PPL 所在的头文件
#include <ppl.h>
#include <iostream>

// 使用 PPL 所在的名字空间 Concurrency
using namespace Concurrency;
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 定义循环次数
    const int num_steps = 10000000;
    // 计算步长
    double step = 1.0/(double) num_steps;

    // 定义一个可合并的 double 数据对象
    // 这样可以在 parallel_for() 算法中并行地对 sums 进行读/写
    // 最后进行合并得到最终的计算结果
```

```

combinable<double> sums;

// 利用 parallel_for() 算法循环计算求和
parallel_for ( 0, num_steps, 1, [&](int i)
{
    // parallel_for() 算法执行的函数体
    double x = (i+0.5)*step;
    // 将结果暂时保存到容器的本地副本中
    sums.local() += 4.0/(1.0+x*x);
} );
// 对可合并对象的结果进行合并, 得到最终结果
double result = sums.combine([](double left, double right) { return left
    + right; });
// 最终得到 PI 的数值
double m_fPI = step * result;

// 输出 PI 的数值
cout<<"PI = "<<m_fPI<<endl;

return 0;
}

```

在这段并行计算 PI 值的程序中, 首先引入了 PPL 所在的头文件和名字空间。接下来的工作就是用 PPL 中的并行算法 `parallel_for()` 对原来的 `for` 循环进行改写。`parallel_for()` 算法会自动根据 CPU 的资源将整个循环划分为不同的范围, 并创建相应的线程并发地执行。

`parallel_for()` 算法因为涉及对同一个共享资源的访问, 所以可以将这个共享资源放到一个可合并的数据对象中。可合并的数据对象会为 `parallel_for()` 算法的每一个线程创建一个本地副本, 这样就可以在 `parallel_for()` 算法中对这个对象的本地副本进行访问, 对其进行读/写操作等。最后将这个对象的各个本地副本合并就得到了最终结果。这比使用临界区进行互斥访问的效率更高。

使用 PPL 中的并行算法, 可以简单快速地将一些算法并行化, 提高程序的运行效率, 从而达到事半功倍的效果。

12.3.4 PPL 中的并行任务

PPL 中的并行算法虽然简单实用, 但是没法对并行的任务进行控制, 比如线程执行的规划、同步、等待和取消等。为了完成这些任务的复杂操作, 就需要使用 PPL 中的并行任务和任务组了。

在 PPL 中, 使用 `task_handle` 类来表示一个任务。一个任务实际上是对一个可以独立完成的工作的封装。这个工作可以用一个函数指针、函数对象或者 Lambda 表达式来表示。任务可以由任何一个线程执行, 特定的任务与特定的线程之间没有绑定关系。通常, 当一个任务需要执行时, 可由系统在当前的线程池中分配一个线程来执行。而负责将任务“分派”到线

程的工作则由运行时来完成。例如，可以使用 Lambda 表达式来简单地创建一个任务：

```
// 使用 Lambda 表达式声明任务
auto taskBoil = make_task( [&]
{
    cout<<"开始烧水"<<endl;
    Sleep(1000);
    cout<<"水烧开了"<<endl;
});
```

这里，使用 `make_task()` 函数简单地创建了一个任务对象，并将它赋值给了 `taskBoil` 变量。注意，这里并没有使用 `task_handle` 来明确指定任务的类型，而是使用了 `auto` 类型来作为任务的类型，编译器会根据 `make_task()` 函数的返回值自动决定任务变量的真正类型，因此省去了推测任务类型的麻烦。

在 PPL 中，一项任务并不能单独执行，如果能够单独执行，那就不是并行计算了。我们必须将任务添加到一定的任务组中，然后通过任务组对这些任务进行管理，按照业务逻辑的需要组织、计划、等待和取消任务等，以使各项任务可以协同工作，最终完成一项比较复杂的功能。例如：

```
// 创建一个任务组
task_group tasks;
// 通过任务组执行任务
tasks.run( taskBoil );
```

有了任务组，就可以对任务进行比较复杂的调度。例如，华罗庚先生在他著名的《运筹方法》中有这样一个经典的任务并行与调度的例子：“比如，想泡壶茶喝。当时的情况是：开水没有；水壶要洗，茶壶、茶杯要洗；火生了，茶叶也有了。怎么办？”

这个问题的最佳解决办法是：洗好水壶，灌上凉水，放在火上；在等待水开的时间里，洗茶壶、洗茶杯、拿茶叶；等水开了，泡茶喝。

这里，我们不去讨论其中的运筹学原理，只探讨如何用 PPL 中的任务和任务组来解决这个烧水泡茶的并行问题。通过分析可以发现，这个问题中有 4 个相对比较独立的工作，可以将这些工作封装到任务中，如下：

```
// 创建函数完成各项任务
// 烧水
void boil(void)
{
    cout<<"开始烧水"<<endl;
    Sleep(5000);           // 任务需要的时间
    cout<<"水烧开了"<<endl;
}
// 洗水壶
void washkettle()
```

```

{
    cout<<"开始洗水壶"<<endl;
    Sleep(1000);
    cout<<"水壶洗完了"<<endl;
}
// 洗茶壶
void washteapot()
{
    cout<<"开始洗茶壶"<<endl;
    Sleep(1000);
    cout<<"茶壶洗完了"<<endl;
}
// 泡茶
void maketea()
{
    cout<<"开始泡茶"<<endl;
    Sleep(1000);
    cout<<"茶泡好了"<<endl;
}

// 主函数
int _tmain(int argc, _TCHAR* argv[])
{
    // 将函数打包成任务
    auto taskBoil = make_task( &boil );
    auto taskWashKettle = make_task( &washkettle );
    auto taskWashTeapot = make_task( &washteapot );
    auto taskMakeTea = make_task( &maketea );

    // ...
}

```

任务创建完成后，接下来的工作就是利用任务组对这些任务进行组织和调度，让这些任务在合适的时候执行，完成烧水泡茶这样一个逻辑过程。

```

// 主函数
int _tmain(int argc, _TCHAR* argv[])
{
    // 创建任务...

    // 利用任务组 task_group 组织任务之间的业务逻辑
    // 使任务在合适的时候执行
    task_group tasks;
    // 首先，执行洗水壶的任务并等待它的完成，
    // 因为它是其他任务的先决条件，只有这项任务
    // 完成后才能执行其他任务
    tasks.run_and_wait( taskWashKettle );

    // 洗水壶的任务完成后，开始并行地执行烧水和洗茶壶的任务
    // 因为这两项任务相对独立，可以在烧水的同时洗茶壶
    // 所以可以用 task_group 类的 run() 函数并发地执行

```

```

tasks.run(taskBoil);
tasks.run(taskWashTeapot);

// 等待烧水和洗茶壶的任务完成
// 使用 task_group 的 wait() 函数, 表示要等到任务组中的
// 所有任务都完成后才能继续往下执行
tasks.wait();

// 烧水和洗茶壶任务完成后, 开始执行最后的泡茶任务,
// 等待泡茶任务完成后, 这个 run_and_wait() 函数才返回
tasks.run_and_wait(taskMakeTea);

return 0;
}

```

现在, 就可以看到程序的输出正确地反映了并行烧水泡茶这样一个比较复杂的业务逻辑。

```

开始洗水壶
水壶洗完了
开始烧水
开始洗茶壶
茶壶洗完了
水烧开了
开始泡茶
茶泡好了

```

现在, 终于可以喝到一杯好茶了, 煮茶论程序, 人生一大快事。

12.3.5 PPL 中的并行对象和并行容器

在并行计算中, 多个线程共享资源的同步与互斥是一个永恒的话题, 如何解决好共享资源的多线程安全访问, 历来是让程序员头疼的一个大问题。为了安全地访问共享资源, 在以往的多线程开发中, 我们必须小心地使用临界区对象或者互斥对象对共享资源的访问同步。一方面, 使用临界区对象或者互斥对象虽然能够解决共享资源的线程安全访问问题, 但是, 共享资源的同步也会带来一定的性能损失。例如, 有多个正在执行的任务要访问某个共享的 vector 容器, 为了保证 vector 容器多线程访问的安全性, 必须在某个线程访问 vector 容器之前锁定这个容器, 这时, 如果有其他线程要访问这个容器, 则只能等待。当这个线程完成 vector 容器的访问之后才会释放这个容器, 这时其他线程才能接着访问这个 vector 容器。多个线程为了访问同一个 vector 容器而进行的等待, 必然带来程序性能的损失。另一方面, 在处理同步的时候, 又要注意对象的锁定与释放, 避免产生线程死锁。这种种因素都使得在编写多线程程序的时候困难重重。

共享资源的多线程安全访问如此困难, 多个线程为了同一个共享资源你争我夺, 这样很不利于创建和谐社会。那么为什么不换一个角度, 将共享资源划分成多个本地副本, 每个线

程在执行的时候，各自独立地访问属于自己的本地副本，再将各个线程的本地副本合并起来形成最终的结果？这样，各个线程都有了属于自己的本地副本，再也不用为了争抢共享资源而你争我夺了，C++世界从此一片安宁祥和。按照这样的思路，PPL 提供了 `combinable<T>` 模板类来解决共享资源的多线程安全访问问题。使用需要共享访问的数据类型实例化这个模板类，就得到了一个新的数据类型，使用这种新类型创建的对象也就是可以多线程安全访问的并行对象。在各个线程执行的时候，可以通过它所提供的 `local()` 函数访问这个对象的本地副本。当多个线程执行完成时，可以通过 `combine()` 函数将多个本地副本合并成我们最终想要的结果。整个过程中，没有共享资源的同步与互斥，没有你争我夺，自然也就提高了对共享资源访问的性能。

并行对象到底好不好，别看广告看疗效。在 11.2.2 小节中，我们曾经利用函数对象统计过 `vector` 容器中所保存的 `Student` 对象的平均身高。现在利用并行对象，这一过程就可以并行执行，且性能很高。

```
// 保存 Student 对象的 vector 容器
vector<Student> vecStu;

// 将 Student 对象保存到 vector 容器中...

// 使用 combinable<T>模板类创建一个 int 类型的并行对象 ncmbTotalH
combinable<int> ncmbTotalH;
// 使用 parallel_for_each() 算法并行地访问 vector 容器，
// 统计 vector 容器中所有 Student 对象的身高总和
parallel_for_each( vecStu.begin(), vecStu.end(),
    [&](Student st)    // 用 Lambda 表达式统计身高总和
{
    // 将 Student 对象的身高累加到并行对象 ncmbTotalH 的本地副本
    ncmbTotalH.local() += st.GetHeight();
});

// 合并并行对象 ncmbTotalH 的本地副本，获得最终结果
int nTotalH = ncmbTotalH.combine(plus<int>());
// 计算并输出平均身高
if( vecStu.size() != 0 )
{
    float fAverageH = nTotalH / vecStu.size();
    cout<<"平均成绩是"<<fAverageH<<endl;
}
```

在这段代码中，首先，`parallel_for_each()` 算法会并行地访问 `vector` 容器中的 `Student` 对象，然后将每个对象的身高累加到并行对象 `ncmbTotalH` 的本地副本；最后，通过并行对象的 `combine()` 函数合并它的多个本地副本就得到了最终的结果。可以看到，各个线程各自独立地访问并行对象的本地副本，这样就避免了共享资源的互斥与同步，程序性能自然也会大大提高，如图 12-5 所示。

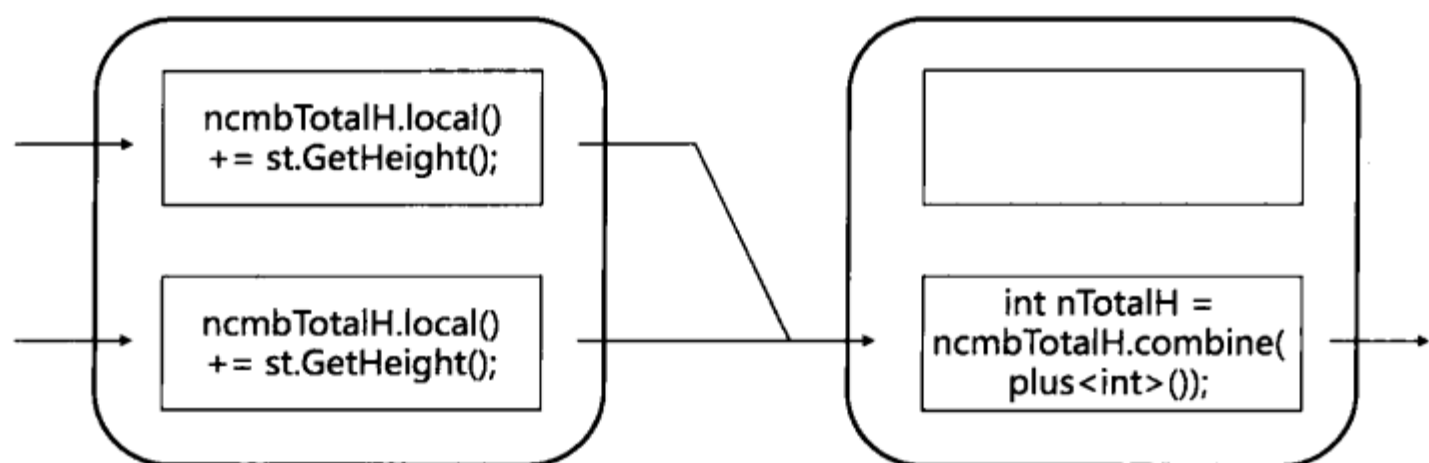


图 12-5 并行对象的执行流程

`combinable<T>`模板类不仅能将基本数据类型包装成并行对象，同样，它也可以将自定义的数据类型包装成并行对象，从而使应用范围更广。一个实际的例子是使用日志系统来记录程序的执行情况。在单线程程序中，程序可以将日志串行地记录到日志系统；对于多线程程序，则可以使用 `combinable<T>`模板类将日志系统包装成并行对象，从而满足多个线程并行访问的要求。

```
// 引入程序所需要的头文件
#include <ppl.h>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>

// 使用标准名字空间 std 和并行名字空间 Concurrency
using namespace std;
using namespace Concurrency;

// 记录每条日志的类
class LogMessage
{
public:
    LogMessage(string strMsg)
    {
        // 保存日志内容
        m_strMsg = strMsg;
        // 将当前时间作为日志时间
        m_tmTime = time(NULL);
    }
    // 获得日志内容和日志时间的接口函数
    string getMsg()
    {
        return m_strMsg;
    }

    time_t getLogTime()
    {
```

```

        return m_tmTime;
    }
    // 因为需要对容器中的日志进行排序,
    // 所以这里重载日志类的 "<" 运算符
    bool operator < (LogMessage m)
    {
        // 以日志的时间先后顺序排序
        return m_tmTime < m.getLogTime() ? true : false;
    }
private:
    // 日志类的属性
    string m_strMsg;        // 日志内容
    time_t m_tmTime;;       // 日志时间
};

// 日志系统类
// 由这个类进行日志的记录和输出
class LogSystem
{
public:
    // 记录日志的成员函数
    void log(string strLog)
    {
        // 将日志保存到容器中
        m_vecMsg.push_back( LogMessage(strLog) );
    }
    // 将另外一个日志系统的日志合并到
    // 当前日志系统的日志中
    LogSystem combine( const LogSystem& other )
    {
        // 获取另外一个日志系统保存日志记录的容器
        vector<LogMessage> omsg = other.getMsg();

        // 合并两个容器
        for( auto it = omsg.begin(); it != omsg.end(); ++it )
            m_vecMsg.push_back( *it );

        // 对合并后的日志记录重新排序
        sort(m_vecMsg.begin(), m_vecMsg.end());

        // 返回合并后的日志系统
        return *this;
    }
    // 获得保存日志记录的 vector 容器
    vector<LogMessage> getMsg() const
    {
        return m_vecMsg;
    }

    // 将日志输出到屏幕和日志文件
    // 输出到屏幕
    void output()

```

```

{
    // 利用 for_each() 算法和 Lambda 表达式,
    // 逐个输出 vector 容器中的日志记录
    for_each( m_vecMsg.begin(), m_vecMsg.end(),
        []( LogMessage msg ){
            time_t tm = msg.getLogTime();
            cout<<ctime( &tm )<<msg.getMsg()<<endl;
        } );
}
// 输出到日志文件
void output(string strFileName)
{
    // 创建并打开日志文件
    ofstream of(strFileName);

    if( of.is_open() )
    {
        // 将日志记录输出到文件
        for_each( m_vecMsg.begin(), m_vecMsg.end(),
            [&]( LogMessage msg ){
                time_t tm = msg.getLogTime();
                of<<ctime( &tm )<<msg.getMsg()<<endl;
            } );
        // 关闭文件
        of.close();
    }
}

private:
    // 保存所有日志记录的 vector 容器
    vector<LogMessage> m_vecMsg;
};

// 用于合并日志系统并行对象的函数对象
class combineLog
{
public:
    // 重载 “()” 操作符, 它接受两个需要合并的日志系统对象,
    // 然后返回合并完成后的日志系统对象
    LogSystem operator () (LogSystem& a, LogSystem& b)
    {
        // 调用日志系统本身的成员函数进行合并
        return a.combine(b);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    // 使用 combinable<T> 模板类包装日志系统类 LogSystem,
    // 创建并行对象 log
    combinable<LogSystem> logsys;
    // 模拟多个线程同时访问并行对象, 记录日志
    auto logthread1 = make_task(

```

```

    [&]{
        // 访问并行对象 log 的本地副本记录日志
        logsys.local().log("线程 1 启动");
        // 执行任务...
        logsys.local().log("线程 1 结束");
    });
    auto logthread2 = make_task(
        [&]{
            logsys.local().log("线程 2 启动");
            // 执行任务...
            logsys.local().log("线程 2 结束");
        });
    // 并行地执行多项任务
    task_group tg;
    tg.run(logthread1);
    tg.run_and_wait(logthread2);

    // 合并并行对象的多个本地副本，得到最终结果
    LogSystem finallog = logsys.combine( combineLog() );
    // 将结果输出到屏幕和日志文件
    finallog.output();
    finallog.output("log.txt");

    return 0;
}

```

在这个完整的例子中，首先创建了一个用于记录日志的日志系统类 LogSystem，它具备基本的记录日志、合并日志及输出日志的功能。为了让程序的多个线程可以并行地访问这个日志系统，使用 combinable<T>模板类将其包装成一个并行对象 logsys。经过这样的包装之后，就可以在多个线程中各自独立地访问这个并行对象的本地副本，通过并行对象 logsys 的本地副本记录日志。最后，通过合并并行对象 logsys 的本地副本，就得到了最终结果 finallog，而最终是把日志还是文件输出到屏幕，就看我们自己的需要了。

虽然并行对象避免了共享资源的互斥与同步，但是它也有一个不足之处，就是所要使用的 combinable<T>模板类包装的共享资源必须是可以合并的，只有可以合并的共享资源才能通过合并多个线程的本地副本而得到最终结果，比如多个线程同时对一个整数进行相加运算，这种结果可以累加合并到最终结果。这样的共享资源，才能使用 combinable<T>模板类包装成并行对象。

除了使用并行对象解决共享资源的互斥与同步之外，很多时候，共享资源都是保存在某个容器中的，这时，容器就成了各个线程争抢的共享资源。为了解决这类问题，PPL 专门提供了两个并行容器——concurrent_vector 和 concurrent_queue。它们的使用跟普通容器的基本相似，只是它们能够保证容器的大部分操作函数都是线程安全的，例如，向 concurrent_vector 容器中添加元素的 push_back() 函数，多个线程可以通过这些线程安全的操作并行地访问容

器。例如：

```
// 引入定义 concurrent_vector 的头文件
#include <concurrent_vector.h>

// ...

// 定义一个可以保存 int 类型数据的并行容器
concurrent_vector<int> cvecInt;

// 使用 parallel_invoke() 并行算法，
// 创建两项任务并行地访问并行容器，分别向并行容器中添加自然数和偶数
parallel_invoke(
    [&] { // 添加自然数
        for(int n=0; n<100; ++n)
            cvecInt.push_back(n);
    },
    [&] { // 添加偶数
        for(int n=0; n<100; ++n)
            cvecInt.push_back(n*2);
    }
);
// 输出并行容器中所保存的数据的数目
cout << "并行容器中保存的数据的数目" << cvecInt.size() << endl;
```

在这里，parallel_invoke()算法的两项任务并行地向并行容器 cvecInt 中添加元素，从而避免了 cvecInt 这一共享资源的互斥与同步。在整个过程中，PPL 保证了这一过程是线程安全的，程序员不需要做任何额外的工作。

不过这里要注意的是，concurrent_vector 容器的实现并不是基于连续内存的，所以不能使用内存地址偏移的方式来访问容器中的元素。例如，通过“&cvecInt[0] + 4”访问的并不是 cvecInt 中的第 5 个元素，而是某个未知位置。小心有地雷啊！

12.3.6 PPL 之外的另一种选择：OpenMP

程序员大约是最懒的一类人。虽然 PPL 可以创建和管理线程，自动完成程序的并行化，但还是要用并行算法或者并行任务对程序进行改写。对于这一点，我们并不满足，有没有一种比 PPL 更简单的方法，无须对程序进行改写而直接将一个串行程序并行化？有！OpenMP (open multi-processing)！

OpenMP 是由 OpenMP Architecture Review Board 牵头提出的、一套用于共享内存并行系统的多线程程序设计的指导性注释 (compiler directive)，目前已被业界广泛接受。OpenMP 支持的程序设计语言包括 C 语言、C++ 语言和 Fortran 语言；OpenMP 支持的编译器包括 Visual Studio 和 Intel Compiler 及开放源代码的 GCC 编译器。OpenMP 提供对并行算法的高层抽象的描述，程序员通过在原始的串行代码中加入专用的 pragma 来指明自己的意图，由此编译器

可以自动将程序进行并行化，并在必要的地方加入同步互斥及通信。当选择忽略这些 pragma 时，或者编译器不支持 OpenMP 时，程序又可退化为通常的程序（一般为串行），代码仍然可以正常运作，只是不能利用多线程来加速程序执行。OpenMP 的使用，可以让程序做到进可攻——在支持 OpenMP 的平台上自动编译成为并行程序，退可守——如果不支持 OpenMP，可以退化为普通的串行程序，从而具有更好的灵活性。

OpenMP 提供的这种并行描述的高层抽象降低了并行编程的难度和复杂度，这样程序员可以把更多的精力投入算法本身，而非其具体的实现细节。对基于并行数据的多线程程序设计，OpenMP 是一个很好的选择。同时，使用 OpenMP 也提供了更好的灵活性，可以较容易适应不同的并行系统配置。线程粒度和负载平衡等是传统多线程程序设计中的难题，但在 OpenMP 中，OpenMP 库从程序员手中接管了部分这两方面的工作。

要想在程序中使用 OpenMP 非常简单，只需要在编译器选项中启用对 OpenMP 的支持，并引入 OpenMP 的头文件 `omp.h` 就可以了。下面看一个简单的例子。

```
#include "stdafx.h"
#include <iostream>

// 引入 OpenMP 的头文件
#include <omp.h>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // 用 pragma 指令指明这是一个可以并行的 for 循环
    // 编译器会根据这些指令自动创建进程，
    // 对 for 循环进行相应的并行处理
    #pragma omp parallel for
    for (int i = 0; i < 10; ++i)
        cout<<i<<endl;

    return 0;
}
```

这是一个简单的输出 0 到 10 整数的程序，使用 OpenMP 之后，这些整数将被并行地输出到屏幕上，不再是从 0 到 10 的依次顺序了。这里可以看到，虽然 OpenMP 的使用跟 PPL 的使用类似，但是比 PPL 的使用更简单。我们只是简单地用一个 pragma 指令告诉编译器接下来的 for 循环是一个可以并行处理的 for 循环，编译器就会根据这些信息自动地创建线程，并并行地执行这个 for 循环，省去了一大堆工作。

更重要的是，还可以通过取消编译器对 OpenMP 的支持，让 pragma 指令失去效用，使程序重新退化成一个串行化的程序。这给程序员提供了极大的灵活性。

找工作就靠它了

C++世界之旅已经告一段落了。现在回想一下，当初拿起这本书开始这段奇妙的 C++ 世界之旅的目的是什么呢？

有的人可能会说是因为学校开了这门课，自己想考一个好成绩；也有的人可能会说是因为自己对编程感兴趣，想学习一门程序设计语言。但是归根结底，大多数人学习 C++ 语言都是为了找一份好的工作。

现在找一份工作很难，特别是找一份关于 C++ 的好工作更难。很多软件公司在招聘员工时都设置了一道门槛——C++ 笔试。只有通过了这道门槛，才有希望进入，找到一份好的工作。经历了这么长时间的学习，游历了 C++ 世界的绮丽风光，现在是将所学知识化为敲门砖、敲开好工作大门的时候了。下面分析了一些著名大公司的 C++ 笔试题目，希望大家能从这些题目中加深对 C++ 语言的理解和积累一些经验，同时能够顺利地通过 C++ 笔试这道门槛。

13.1 打好基础

万丈高楼平地起。任何时候，基础知识都是最重要的。基础知识掌握得牢固与否，已成为衡量一个程序员水平高低的重要标准。只有基础牢固了，程序员才可能在工作中减少失误，做出更好的成绩。既然基础知识这么重要，当然它也成了 C++ 笔试题最重要的部分。

13.1.1 基本概念

题 1 运算符与表达式。

判断 (A)、(B)、(C)、(D) 四个表达式是否正确，若正确，写出经过表达式运算后 a 的值。

```
int a = 4;  
(A) a += (a++);  
(B) a += (++a);  
(C) (a++) += a;  
(D) (++a) += (a++);
```

【分析】 这道题考查的是 C++ 语言中的“++”运算符及表达式的相关知识，只要复习本书中关于“++”运算符及表达式的相关介绍，就可以轻松回答这个问题。其中，(C) 表

达式错误，因为赋值运算符左侧不是一个有效变量，不能赋值，可改为 $(++a) += a$ 。修改后 a 的值依次为 9、10、10、11。

题 2 C++中的模板类有什么优势？

【分析】 这道题考查的是对 C++语言中模板类的认识和理解。根据在实践中应用模板类的体会，可以将模板类的优势总结如下：

- 可用来创建动态增加或减少的数据结构；
- 它是类型无关的，因此具有很高的可复用性；
- 它在编译时而不是运行时检查数据类型，保证了类型的安全；
- 它是平台无关的，具有很好的可移植性。

题 3 什么是“引用”？声明和使用“引用”时要注意哪些问题？

【分析】 这道题考查对“引用”这个基本概念的理解。实际上，在本书中已经分析了引用的实质就是变量的“绰号”，按照这个思路去回答问题就可以了。

专业地讲，引用就是某个目标变量的“别名（alias）”，对引用的操作与对变量的直接操作效果完全相同。声明一个引用时，切记要对其进行初始化，将其与某个目标变量相关联。引用声明完毕后，相当于目标变量名有两个名称，即该目标的原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

题 4 将“引用”作为函数的参数有哪些特点？

【分析】 这道题不仅考查了对引用的理解和应用，同时考查了对函数的三种传递参数方式基本概念的理解。总结起来，使用“引用”作为函数的参数有如下特点。

- 传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就被当成原来主调函数中实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应目标对象（在主调函数中）的操作。
- 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还要调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率更高，所占空间也更少。

- 使用指针作为函数的参数虽然也能达到跟使用引用相同的效果，但是，一方面，在被调函数中同样要给形参分配存储单元，在使用时需要重复使用以“*指针变量名”的形式进行运算，这很容易产生错误且程序的可读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。因而引用更容易使用，更清晰。

题 5 给引用赋值意味着什么？

【分析】 这道题虽然表面上考查的是引用的赋值，但实际上，如果能回答给引用赋值时应该注意的问题，那会给自己加分不少。

引用就是另外一个对象的别名，给引用赋值，就是将引用指向这个对象。此后对引用的任何操作，实际上就是对这个对象的操作。

记住：引用就是目标对象的指示物。所以当定义一个引用时，必须给引用赋值，将它指向某个对象。并且引用和它所指向的对象之间的这种指向关系一旦确定就不能更改，也就是说，不能修改引用的初始指向，将其指向另外一个对象，引用和它所指向的对象之间的关系是从一而终的。

题 6 如何将数字转换成对应的字符串？

【分析】 在开发实践中，常常需要将数字转换成相应的字符串，以达到某些特殊的目的。例如，可以通过将浮点数转换成字符串来进行两个浮点数的比较。虽然 C++ 语言提供了很多函数，比如 `_itoa()`、`_itow()` 及 `_ltow()` 等，但是更多时候还是使用 `stringstream` 类来完成数字到字符串的转换。例如：

```
// 引入定义 stringstream 的头文件
#include <sstream>
#include <iomanip>

// ...
// 需要比较的两个浮点数大小
float fA = 1.19830706;
float fB = 1.19830703;
// 定义一个 stringstream 对象
stringstream sstr;

// 设定浮点数的输出精度，并将 fA 输出到 stringstream 对象
sstr<<fixed<<setprecision(8)<<fA;
string strA;
// 将浮点数 fA 转换为字符串 strA
sstr>>strA;
// 清空 stringstream 对象，开始第二次转换
sstr.clear();
sstr<<fB;
string strB;
sstr>>strB;
```

```
// 通过转换后的字符串比较两个浮点数的大小
if( strA == strB )
    cout<<fA<<"等于"<<fB<<endl;
```

除了将数字直接转换为对应的字符串外，还可以在转换的过程中对数字的输出格式进行控制，将数字转换为八进制或者十六进制等形式的字符串，以满足不同的需要。stringstream 类除了可以将数字转换为字符串外，相反，它也可以将字符串转换为数字。例如：

```
stringstream sstr;
// 定义一个需要转换的表示数字的字符串 strA
string strA("1.1983");
// 将字符串输出到 stringstream 对象
sstr<<strA;
// 将字符串 strA 转换为浮点数 fA
float fA = 0;
sstr>>fA;
```

题 7 什么是预编译？

【分析】 这道题考查对编译器知识的理解。编译器是编写程序的关键工具，当然了解越多越好。预编译又称为预处理，它完成的是代码文本的替换工作。在进行正式的编译之前，预编译程序会处理代码中以“#”开头的指令，比如拷贝“#include”包含的文件代码、“#define”宏定义的替换、条件编译等，都是为编译做的一些预备工作。

13.1.2 函数

题 8 请编写一个函数将一个链表反转。例如，现在有一个链表 1->2->3->4->5，通过调用函数将链表反转成为 5->4->3->2->1。

【分析】 这道题不仅考查对数据结构的理解，而且考查对编写函数基本功的掌握情况。单向链表的反转既是最常见的笔试题，也是一个非常基础的问题。要回答这个问题，首先最容易想到的方法就是遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素；然后将当前节点元素的指针反转后，利用已经存储的指针往后继续遍历。完整的代码如下：

```
// 定义链表数据结构
struct link {
    int data;           // 当前节点的数据
    link* next;        // 指向下一个节点的指针
};

// 反转链表函数
void reverse(link*& head)
{
    // 对输入参数的有效性进行验证
    // 这是一个很好的编程习惯，将为你在考官面前加分不少
```

```

    if(head ==NULL)
        return;
    // 定义前一节点、当前节点和后一节点
    link *pre, *cur, *next;
    pre = head;
    cur = head->next;
    while(cur)
    {
        next = cur->next;
        cur->next = pre;
        pre = cur;
        cur = next;
    }
    head->next = NULL;
    head = pre;
}

```

当然，解决问题的方法不止一个。这道题还可以利用递归的方法解决。这种方法的基本思想是在反转当前节点之前，先调用递归函数反转后续节点。不过这种方法有一个缺点，就是在反转后的最后一个节点会形成一个环，所以必须将函数返回的节点的 next 预置为 NULL。因为要改变 head 指针，所以这里需要使用引用。算法的源代码如下：

```

// 通过递归反转链表
link* reverse(link* p, link*& head)
{
    // 判断是否满足返回条件
    if( p == NULL || p->next == NULL)
    {
        head = p;
        return p;
    }
    else
    {
        link* tmp = reverse(p->next, head);
        tmp->next = p;
        return p;
    }
}

```

题 9 请编写一个字符串拷贝函数。

【分析】乍一看这个题目可能会认为字符串拷贝还不简单吗？逐个复制字符串数组中的元素就可以了啊。也许，你可以很快给出如下解答：

```

// 糟糕的字符串拷贝函数
void strcpy( char *strDest, char *strSrc )
{
    // 逐个复制字符串数组中的数据，直到字符串结束
    while( (*strDest++ = * strSrc++) != '\0' );
}

```

如果你将这个答案交给考官，也许考官会客气地让你回去等消息，但是可以肯定地告诉

你，你永远也不会等到消息。实际上，别看这道题简单，但是它却暗含杀机，处处危机四伏。看看下面的解答，就清楚自己到底失误在什么地方了。比较完美的解答如下：

```
// 为了实现链式操作，将目标地址返回
char * strcpy( char *strDest,
               const char *strSrc ) // 将源字符串加 const 修饰，表明其为输入参数
{
    // 对源地址和目的地址加非 0 断言，判断其有效性
    assert( (strDest != NULL) && (strSrc != NULL) );
    // 保存目标地址返回
    char *address = strDest;
    // 逐个复制字符串数组中的数据，直到字符串结束
    while( (*strDest++ = *strSrc++) != '\0' );

    // 返回目标地址
    return address;
}
```

从糟糕的 strcpy() 函数到完美的 strcpy() 函数，考查了在编写函数解决问题时考虑的问题是否全面。只有具备了扎实的基本功，才能够写出完美的 strcpy() 函数，才是软件公司要找的人。

题 10 内联函数有什么意义？

【分析】 这道题考查对内联函数的理解和使用。当编译器内联展开一个函数调用时，内联函数的代码会被插入调用代码流中，这样程序优化器就能够顺序集成被调用代码，也就是将被调用代码直接优化进调用代码中，避免了函数调用，从而改善程序的性能。

在 C++ 语言中，有好几种方法可以将一个函数设定为内联函数。其中一些需要使用 inline 关键字，还有一些则不需要，例如直接在类定义中定义成员函数，这个成员函数就被设定为内联函数。不管使用何种方法设定函数为内联函数，这只是给编译器一个“建议”，编译器可以忽略它。编译器可能会展开内联函数调用，也可能不展开，最终的决定权还是在编译器的手上。这就给了编译器很大的灵活性，编译器能够区别对待很长的函数和简短的函数。另外，如果选择了正确的编译选项，编译器还能生成易于调试的代码。

13.1.3 面向对象思想

题 11 面向对象思想有什么用？

【分析】 面向对象技术是开发大型的、复杂的应用软件和系统的最佳方法，了解它的作用和意义，有助于学习和应用面向对象技术。

在处理大型的、复杂的软件系统方面，软件工业是“失败的”。但是这种“失败”实际上归因于我们的“成功”：我们的“成功”使得用户想得到更多、更复杂的软件。不幸的是，

创造了市场的饥渴，而“结构化”的分析、设计和编程技术却无法满足这种饥渴。因此需要创造一种更好的程序设计思想，这就是面向对象程序设计思想。

C++语言既可以支持面向对象编程，也可以当成传统的编程语言使用（作为“一种更好的 C 语言”），或者泛型编程使用。每种方法都有其优点和缺点，不要期望在使用一种方法时能得到另外一种技术的好处。如果把 C++语言作为“一种更好的 C 语言”来使用，那么不要期望能得到面向对象所带来的好处。鱼与熊掌是不可兼得的。

题 12 面向对象思想的三个基本特征是什么？

【分析】 对面向对象思想的三个基本特征的理解，是掌握面向对象技术最基本的要求，任何一个合格的程序员都应该能够回答这个问题。面向对象思想的三个基本特征如下。

- 封装。封装将客观事物抽象成类，每个类对自身的数据和方法实行访问控制，通过关键字（private、protected 和 public）来控制外界对类成员的访问，以达到保护数据和方法的目的。
- 继承。子类从父类继承，从而获得父类的属性和方法。广义的继承有三种实现形式：实现继承（指使用基类的属性和方法而无须额外的编码能力）、可视继承（子窗体使用父窗体的外观和实现代码）、接口继承（仅使用属性和方法，实现滞后到子类实现）。前两种（类继承）和后一种（对象组合成接口继承及纯虚函数）构成功能复用的两种方式。
- 多态。多态将父对象设置成和一个或更多的它的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单来说就是一句话：允许将子类类型的指针赋值给父类类型的指针。

题 13 何时应该使用继承？

【分析】 这道题进一步考察继承机制的运用。在现实世界中，抽象事物有两种角度：“种类”和“部分”。例如，Student st 是一种（is-a-kind-of-a）学生（Student），并且 Student st 有（has-a）手（Hand）和脚（Foot）等。两者之间的“部分”关系可以通过类的成员属性来表达，而继承则用来表达两者之间的“种类”关系。当某个事物是另一个事物的“一种”时，就应该使用继承。

13.1.4 类与对象

题 14 类是什么？对象又是什么？

【分析】 在 C++语言中，类与对象是面向对象思想的承载者，是面向对象思想在程序代

码中的具体体现。所以，类与对象同样成为 C++ 笔试的一个重点和难点。类与对象是面向对象编程（OOP）中两个最基本的概念，但是，越基本的概念越能考察我们的基本功到底如何。

简单地讲，类是对现实世界中同一类物体的抽象，可以用来定义表达这类物体的数据类型。从计算机科学的角度来理解，类由表达物体状态的成员属性和转换这些状态的成员函数组成。例如，int 既有状态集合，也有像 $i+j$ 或 $i++$ 等这样的操作，所以 int 也是一种类。同样，类提供了一组操作集合（通常是 public 访问控制类型）和一组描述类型实例所拥有的抽象值的数据集合。

既然类是同一类物体的抽象，那么类的对象就可以看成是这类物体的某个实例了。我们声明了“int i”，就可以说“i 是 int 类型的一个对象”。在 OO/C++ 中，“对象”通常意味着“类的一个实例”。因此，类定义多个对象（实例）的可能的行为，而对象则将类具体化、实例化。

题 15 请简要描述 struct 和 class 的区别。

【分析】 这道题考查面向对象中 struct 和 class 这两个关键字在构建类时的区别。实际上，struct 和 class 在语法上的区别很小，它们两个唯一的区别就是默认情况下成员的访问权限不同，struct 的成员默认是公有的，而 class 的成员默认是私有的。

如果这个问题只回答到这里，只能算回答了一半。从感觉上讲，大多数程序员觉得类和结构有很大的区别，觉得结构仅仅像一堆缺乏封装和功能的开放的内存位，而类就像活的并且可靠的社会成员，它有智能服务、有牢固的封装屏障和一个定义良好的接口。既然大多数人都这么认为，那么只有当类有很少的方法并且有公有数据（这种事情在良好的设计系统中是存在的）时，才能使用 struct 关键字；否则，应该使用 class 关键字。这些恐怕才是考官想要看到的答案。

题 16 重载（overload）和重写（override）的区别是什么？

【分析】 这是一个常考的题目。面向对象中的重载和重写，让类具备了千变万化的能力，这也是最让程序员着迷的地方。

从定义上来说，两者的区别如下。

- 重载。重载是指允许同时存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。
- 重写。重写是指子类重新定义父类虚函数的方法。

从实现原理上来说，两者的区别如下。

- 重载。重载编译器根据函数声明中的不同的参数表，对同名函数的名称做修饰，然

后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。例如，有两个同名函数——`void func(int n)`和 `void func(string str)`，虽然这两个函数的名称都是 `func`，但是经过编译器修饰后的函数名称可能是这样的：`int_func` 和 `str_func`。对于这两个函数的调用，在编译期间就已经确定了，它们是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关。

- 重写。重写和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同子类指针动态地调用属于子类的该函数，这样的函数调用在编译期是无法确定的（无法给出调用子类的虚函数的地址）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

题 17 子类覆盖父类的虚函数是否不用加 `virtual` 关键字？

【分析】 `virtual` 修饰符是会隐性继承的，派生类中的 `virtual` 关键字可加可不加。子类的空间里有父类的所有变量（`static` 除外）。同一个函数只存在一个实体（`inline` 除外）。子类覆盖其函数不加 `virtual` 关键字也能实现多态。但是，为了增加代码的可读性，最好还是在子类中的虚函数前添加 `virtual` 关键字。

题 18 能重载类的析构函数吗？

【分析】 这道题考查对类的析构函数的认识。对于析构函数，不仅应该认识到它不能重载，而是应该认识到它为什么不能重载。在 C++ 语言中，类有且只能有一个析构函数。例如，`DemoClass` 类的析构函数只能是 `DemoClass::~~DemoClass()`，不带任何参数，也不返回任何东西甚至 `void`。由于永远不会显式地调用析构函数，因此无论何时都不能传递参数给析构函数，也就无法根据参数的变化来形成析构函数的重载。

题 19 局部对象析构的顺序是什么？

【分析】 这道题考查 C++ 语言中一个鲜为人知的知识点。在大多数时候，虽然局部对象的析构顺序并不用我们操心，但是在某些特殊情况下，清楚认识局部对象的析构顺序，可以帮助我们正确地析构局部对象。

局部对象的析构与它们的创建是反序的，也就是说，先被构造对象，后被析构。例如，在下面的代码中，局部对象 `a` 先于局部对象 `b` 创建，但是却后于 `b` 析构。

```
void func()
{
    DemoClass a;
    DemoClass b;
    // ...
}
```

题 20 在派生类的析构函数中，需要显式调用基类的析构函数吗？

【分析】 这道题仍旧考查对析构函数的认识。因为 C++ 语言中类的析构函数往往涉及内存资源的清理和释放，稍有不慎就有可能造成令人头疼的内存泄露问题，所以析构函数对于一个类的编写尤其重要。对于析构函数的认识和理解，自然也就反映了我们的 C++ 语言功力到底有多深。

回到问题本身，永远不需要显式调用析构函数，在派生类的析构函数中，自然也不需要调用基类的析构函数。派生类的析构函数（不论我们是否显式地定义了）会自动调用基类的析构函数，并且基类的析构函数是在派生类的成员对象被析构之后调用的。例如：

```
// 成员对象的类
class Member
{
public:
    ~Member();
    // ...
};

// 基类
class Base
{
public:
    virtual ~Base();    // 基类的虚析构函数
    // ...
};

// 派生类
class Derived : public Base
{
public:
    ~Derived();          // 派生类的析构函数
    // ...
private:
    Member m_Member;
};

// 定义派生类的析构函数
Derived::~~Derived()
{
    // 先析构成员对象，然后析构基类
    // 编译器自动调用 m_Member.~Member()
    // 编译器自动调用 Base::~~Base()
}
```

题 21 可以将一个派生类指针转换成它的基类指针吗？

【分析】 在开发实践中，常常要进行指针类型的转换，其中最常见的情况就是将一个派生类指针转换为一个基类指针，也就是用一个基类指针来指向一个派生类的对象。派生类对象是基类对象的一种。因此，从派生类指针到基类指针的转换是非常安全的，并且始终可以转换成功。例如，如果有一个基类 Human 类型的指针，而实际上指向了一个派生类 Student 对象，这种从 Human* 到 Student* 的转换是非常安全的和常规的。例如：

```

// 基类
class Human
{
public:
    // 成员函数
    string getName()
    {
        return m_strName;
    }
private:
    // 成员属性
    string m_strName;
};
// 派生类
class Student : public Human
{
    // ...
};
// 一个接受基类 Human 类型指针的函数
string getName(Human* pHuman)
{
    // 调用基类的成员函数
    return pHuman->getName();
}
string getStudentName(Student* pStu)
{
    // 不用进行显式的类型转换
    // 派生类 Student 的指针将被隐式地、安全地转换成基类指针
    // 直接使用派生类 Student 类型的指针作为参数,
    // 调用需要基类 Human 类型指针为参数的函数
    return getName(pStu);
}

```

题 22 C++是如何实现指针变量的静态类型和动态绑定的?

【分析】 我们知道, C++是一种强类型的编程语言, 它的每一个变量都有其确定的类型。但是, 当这个变量是指针的时候, 情况就有些特殊。我们有一个某种类型的指针, 而它实际上指向的是其派生类的某个对象。例如, 一个 Human* 指针实际上指向一个 Student 对象。

```

Student st;
Human* pHuman = &st; // 将基类指针指向派生类对象

```

这时, pHuman 指针就有了两种类型: 指针的静态类型 (在此是 Human) 和它所指向的对象的动态类型 (在此是 Student)。

静态类型意味着成员函数调用的合法性将被尽可能早地检查: 编译器在编译时, 编译器可以使用指针的静态类型来决定成员函数的调用是否合法。如果指针类型能够处理成员函数, 那么指针实际上指向的对象当然能很好地处理它。例如, 如果 Human 类有某个成员函数, 而指针实际上指向的 Student 对象是 Human 的一种, 是 Human 类的派生类, 自然会从基类

Human 中继承获得相应的成员函数。

动态绑定意味着在编译时，通过指针实现成员函数调用的代码地址并没有确定，而在运行时，这一地址才根据指针所指向的实际对象的真正类型动态地决定。因为绑定到实际被调用的代码的这个过程是在运行时动态完成的，所以被称为“动态绑定”。动态绑定是虚函数所带来的 C++ 特性之一。下面的代码演示了 C++ 语言中的静态类型和动态绑定。

```
// 基类
class Human
{
public:
    Human(string strName)
        : m_strName(strName)
    {}
    // 成员函数
    string getName()
    {
        return m_strName;
    }
    // 虚成员函数
    virtual string getJobTitle()
    {
        return "";
    }
private:
    string m_strName;
};

// 派生类
class Student : public Human
{
public:
    Student(string strName)
        : Human(strName)
    {}
    // 重新定义虚成员函数
    virtual string getJobTitle()
    {
        return "Student";
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    Student st("Jiawei");
    Human* pHuman = &st; // 基类指针指向派生类对象
    // 调用 pHuman 指针所指向对象的 getName() 普通成员函数
    // 编译时，首先根据 pHuman 的静态类型 Human 检查成员函数调用的合法性
    cout<<"姓名: "<<pHuman->getName()<<endl;

    // 如果调用的是虚成员函数，
    // 将在运行时根据 pHuman 所指向的真正对象决定此成员函数的地址。
```

```

// 虽然 pHuman 的静态类型是 Human，但它所指向的实际上是一个 Student 对象，
// 所以这里的成员函数将不再是调用 Human 类的 Human::getJobTitle()，
// 而是被动态绑定为 Student 类的 Student::getJobTitle()，
// 输出 Student 对象的真实情况
cout<<"职位: "<<pHuman->getJobTitle()<<endl;

return 0;
}

```

题 23 在 C++ 语言中，如何实现接口和实现的分离？

【分析】 在 C++ 语言中，接口是一个程序最有价值的资源。设计良好的接口，可以降低整个系统的耦合性，提高系统的内聚，从而达到面向对象“高内聚，低耦合”的设计目的。同时可以让类更易于使用，更具可维护性。但是，接口的设计往往比用一堆类来实现这个接口更费时间。既然接口如此有价值，它们应该被保护起来，以免因为数据结构和实现的改变而被破坏。因此，应该将接口和具体的实现分离，而在 C++ 语言中，我们是通过抽象基类（abstract base class）来实现接口和实现相分离的。

在设计层次上，抽象基类表达的是一种抽象概念。例如，我们说“动物都会动”，这里的“动物”并不是特指某一种动物，也不是某一个动物，而是表达的一种抽象概念。因此这里的“动物”就是抽象基类。在 C++ 语言中，我们用带有一个或者多个纯虚成员函数的类来表示一个抽象基类。例如：

```

// 动物抽象基类
class Animal
{
public:
    // 使用纯虚函数表示接口
    virtual void Move() = 0; // = 0 表示这是一个纯虚成员函数
};

```

在 Animal 类中，我们使用了一个纯虚函数 Move() 来表达这个抽象基类的接口。因为抽象基类只是用来表达“接口”这个抽象概念的，带有纯虚函数，所以它并不能实例化，也就是说，不能创建某个 Animal 类型的对象。要创建某个具体的动物，必须从抽象基类派生具体的派生类，实现抽象基类中的纯虚成员函数，也就是将接口逐个实现。

```

// 从抽象基类派生新的类，实现接口
class Bird : public Animal
{
public:
    // 实现抽象基类的纯虚成员函数，也就是实现接口
    virtual void Move()
    {
        cout<<"鸟儿飞"<<endl;
    }
};

```

```

// 从抽象基类派生另外一个类，接口的第二种实现
class Horse : public Animal
{
public:
    // 实现抽象基类的纯虚函数
    virtual void Move()
    {
        cout<<"马儿跑"<<endl;
    }
};

// ...
// 创建保存具体实现对象的容器
// 为了使用动态绑定，避免对象的释放，这里使用了智能指针
vector<shared_ptr<Animal>> vecAnimal;
// 创建实现类的具体对象，并保存到容器中
vecAnimal.push_back(shared_ptr<Animal>(new Bird));
vecAnimal.push_back(shared_ptr<Animal>(new Horse));
// 访问容器中的实现类的对象
for_each( vecAnimal.begin(), vecAnimal.end(),
    [](shared_ptr<Animal> spAnimal)
    {
        // 调用抽象类提供的接口
        // 在运行时将根据动态绑定原则，调用具体的实现类对接口的实现
        spAnimal->Move();
    });

```

在这里，使用抽象基类 Animal 类来表达接口，用从抽象基类 Animal 类派生的 Bird 类和 Horse 类来表达对接口的不同实现。这样，就可以将接口在抽象基类中固定下来，同时又可以在实现类中灵活地修改接口的实现。例如，如果有一天这匹马变成了会飞的马，就可以将 Horse 类的实现修改如下：

```

// 修改实现类的具体实现，变成会飞的马
class Horse : public Animal
{
public:
    // 修改接口的实现
    virtual void Move()
    {
        cout<<"马儿飞"<<endl;
    }
};

```

这样，表示接口的抽象基类可以保持不变，只需要修改具体的实现，就轻松满足了需求的变化。这样也很好达到了接口和实现互不干扰、相互分离的目的。

题 24 如何将自定义的类通过标准输出流对象输出？

【分析】 C++语言中的基本数据类型，比如 int、string 等，都可以通过标准输出流对象 cout 直接输出到屏幕，这是因为标准库重载了可以接受 int 对象的“<<”运算符。同理，通

过重载“<<”运算符,自定义的类也可以通过标准输出流对象直接输出。至于输入运算符“>>”,也是同样的道理。例如:

```
// 在需要输入/输出的类中定义“<<”和“>>”运算符为友元
// 这样,在这两个运算符中就可以直接访问这个类的成员属性
class Student
{
// 声明“<<”和“>>”运算符为友元
    friend ostream& operator<< (ostream& o, const Student& st);
    friend istream& operator>> (istream& i, Student& st);
    // ...

private:
    string m_strName;        // 需要输入/输出的成员属性
};

// 重载“<<”运算符,实现 Student 类的输出
ostream& operator<< (ostream& o, const Student& st)
{
    // 输出 Student 对象的成员属性
    return o << st.m_strName;
}

// 重载“>>”运算符,实现 Student 类的输入
istream& operator>> (istream& i, Student& st)
{
    // 输入 Student 对象的成员属性
    return i >> st.m_strName;
}

int _tmain(int argc, _TCHAR* argv[])
{
    Student stChen;
    // 从屏幕输入 Student 对象
    cin>>stChen;
    // 将 Student 对象输出到屏幕
    cout<<stChen<<endl;
    return 0;
}
```

通过重载“<<”和“>>”运算符,实现了自定义类与屏幕之间的直接输入与输出。更进一步,还可以通过重载这两个运算符,实现类与文件之间的直接输入与输出,通过这样的重载,类的读取与保存将更加方便直接。例如,我们在 12.3.5 小节中介绍过如何将 ThreadLog 类中所保存的 LogMessage 对象输出到文件。现在,利用“<<”运算符重载和 STL 中的 copy() 算法,这一过程将更加简单方便。

```
// 声明运算符“<<”为 LogMessage 类的友元
class LogMessage
{
    friend ostream& operator << (ostream& o, const LogMessage& msg);
    // ...
};
```

```

// 重载 "<<" 运算符，实现 LogMessage 类的输出
ostream& operator<< (ostream& o, const LogMessage& msg)
{
    // 输出 LogMessage 对象的成员属性
    time_t tm = msg.m_tmTime;
    o<<ctime( &tm )<<msg.m_strMsg<<endl;

    // 返回输出流对象
    return o;
}

```

重载接受 LogMessage 对象的 "<<" 运算符之后，输出 ThreadLog 类中的日志将更加简单。

```

class ThreadLog
{
// ...
// 将 ThreadLog 中保存的所有日志输出到文件
void output(string strFileName)
{
    ofstream of(strFileName);

    if( of.is_open() )
    {
        // 使用 copy() 算法及重载之后的 "<<" 运算符，
        // 将 m_vecMsg 容器中的所有日志记录 LogMessage 对象输出到文件
        copy(m_vecMsg.begin(), m_vecMsg.end(),
            ostream_iterator<LogMessage>(of, "" ));

        of.close();
    }
};

```

题 25 完成自己的 String 类。

已知 String 类的定义如下：

```

class String
{
public:
    String(const char *str = NULL);           // 通用构造函数
    String(const String &another);           // 拷贝构造函数
    ~String();                                // 析构函数
    String & operator =(const String &rhs);  // 赋值函数
private:
    char *m_data;                             // 用于保存字符串
};

```

请尝试写出类的成员函数的实现。

【分析】 这道题几乎考查了面向对象的方方面面，从构造函数到析构函数，从操作符

重载到类成员的实现，所以这道题成为考官的常用考题。如果理解了面向对象和类的相关知识，就可以给出完美的答案。

```
// 根据字符串指针构造 String 对象
String::String(const char *str)
{
    // strlen()函数在参数为 NULL 时会抛出异常，所以要进行参数的判断
    if ( str == NULL )
    {
        // 如果传入的参数为 NULL，则构造一个空字符串
        m_data = new char[1] ;
        m_data[0] = '\0' ;
    }
    else
    {
        // 根据传入的字符串长度申请内存
        m_data = new char[strlen(str) + 1];
        // 复制字符串
        strcpy(m_data, str);
    }
}

// 根据传入的 String 对象构造新的 String 对象
String::String(const String &other)
{
    // 申请内存
    m_data = new char[strlen(other.m_data) + 1];
    // 复制字符串
    strcpy(m_data, other.m_data);
}

// 重载操作符“=”，进行字符串的赋值操作
String& String::operator =(const String &rhs)
{
    // 判断是否是自己给自己赋值，如果是，则直接返回
    // 代码中是否有这个判断，是程序员是否有足够经验的标志
    if ( this == &rhs)
        return *this ;
    // 删除原有数据
    delete []m_data;
    // 为新的数据申请内存
    m_data = new char[strlen(rhs.m_data) + 1];
    // 复制字符串
    strcpy(m_data, rhs.m_data);

    return *this ;
}

// 析构函数，清理资源
String::~~String()
{
    // 释放字符串
}
```

```
        delete []m_data ;
    }
```

13.1.5 STL

题 26 如何删除容器中的元素？

某冒牌程序员写了如下这段代码来删除容器中所有符合条件的数据，请你帮这个程序员看看这段代码有什么问题？

```
typedef vector<int> IntArray;
IntArray Array;
Array.push_back( 1 );
Array.push_back( 2 );
Array.push_back( 2 );
Array.push_back( 3 );
// 删除 Array 数组中所有的数字 2
for( IntArray::iterator itor = Array.begin();
    itor != Array.end(); ++itor )
{
    if( 2 == *itor )
        Array.erase( itor );
}
```

【分析】 这道题考查了 vector 容器的使用。仔细分析代码，我们会发现代码有两个问题。首先是代码在使用 typedef 定义新的容器类型时，缺少容器的类型参数。在这里，应该以“vector<int>”的形式指明 vector 容器的类型参数为 int。其次是 vector 容器删除元素的问题，如果认真阅读了本书，这个问题一定早就发现了，因为在本书中关于 vector 的部分已经提示过这个问题。每次代码在调用“Array.erase(itor);”时，被删除元素之后的内容会自动往前移，这将导致迭代漏项，应在删除一项后进行“itor++”运算，使之从已经前移的下一个元素起继续遍历。实际上，可以使用 remove() 算法结合 vector 容器的 erase() 操作函数轻松地完成这一任务。

```
// 删除 Array 数组中所有的数字 2
Array.erase(remove(Array.begin(), Array.end(), 2), Array.end());
```

remove() 算法会将容器中所有符合条件的数据移动到容器的末尾，并返回表示容器中所有剩下数据的新的结束位置的迭代器，而这个迭代器又可以作为 erase() 函数的参数，用来擦除容器末尾中符合删除条件的数据。

题 27 灵活运用 STL 算法，由繁至简解决实际问题。

一个班级的学生成绩保存在 vecScore 容器中，请统计出其中的及格人数。

【分析】 虽然这是一道简单的对容器中的数据进行统计的题目，但是也能够考查我们对 STL 的熟练运用程度。如果对 STL 只有一个初步的了解，知道如何使用迭代器访问容器中的

数据，那么实现可能是这样的：

```
// 保存学生成绩的容器
vector<int> vecScore;
// 将学生成绩数据保存到容器
// ...
// 保存及格人数的变量
int nPass = 0;
// 使用 for 循环遍历容器，用迭代器访问容器中的数据，逐个进行判断
for(auto it = vecScore.begin(); it != vecScore.end(); ++it )
{
    if( *it >= 60 )
        nPass += 1;
}
cout<<"及格人数是"<<nPass<<endl;
```

如果这道题的满分是 10 分，上面这种使用 for 循环的实现方法最多只能拿到 3 分，还没有及格呢。在 C++ 语言中，解决问题的方法永远不止一个。当我们对 STL 有了进一步的了解之后，实现可能如下：

```
int nPass = 0;
// 使用 for_each() 算法结合 Lambda 表达式统计容器中的数据
for_each(vecScore.begin(), vecScore.end(),
    [&](int nScore)    // 完成统计的 Lambda 表达式
    {
        if(nScore >= 60)
            nPass +=1;
    });
cout<<"及格人数是"<<nPass<<endl;
```

在第二种实现方法中，综合运用了 for_each() 算法和 Lambda 表达式来完成数据的统计，整个过程比 for 循环更加简洁流畅。这样的实现方法应该可以拿个满分吧？可惜的是，考官只给了一个及格分数 6 分。虽然 for_each() 算法避免了 for 循环的烦琐，但是仍然需要使用 Lambda 表达式逐个判断容器中的数据来完成统计，这就是为什么这种实现方法只能拿到一个及格分数。那么，剩下的 4 分该如何拿到手呢？这就要靠我们对 STL 有更深入的理解和熟练的运用。当对 STL 的算法部分有了更深入的理解之后，只需要一行代码就可以完成该统计任务：

```
// 使用 count_if() 算法和函数对象统计容器中的数据
int nPass = count_if( vecScore.begin(), vecScore.end(),
    not1(bind1st(greater<int>(), 60)));
```

在第三种实现方法中，综合运用了 count_if() 算法、greater<int> 函数对象，以及两个辅助函数 not1() 和 bind1st()，在短短的一行代码中就完成了数据的统计任务，避免了循环遍历容器中的数据，也避免了逐个判断数据的大小来完成统计。这样的实现方法才是真正将 STL 运用到了极致，发挥了它简洁优雅的特点，这才是让考官满意的满分答案。

13.2 积累经验

程序员为什么越老越值钱？值钱的是他的经验。如果你能够在考官面前表现出是一个经验丰富的编程老手，那么恭喜你，这份 Offer 是你的了。

题 28 const 与 #define 相比，请说出它们各自有何优点？

【分析】 const 和 #define 的区别，已经在本书中作了详尽的解释，总结起来，const 的优势如下。

- const 常量有数据类型，而 #define 的常量没有数据类型。编译器可以对前者进行类型安全检查；对后者只能进行字符替换，没有类型安全检查，并且字符替换可能会产生意想不到的错误。
- 有些集成化的调试工具可以对 const 常量进行调试，但是不能对 #define 的常量进行调试。

题 29 下面是两种 if 语句判断方式。请问哪种写法更好？

```
int n;  
if (n == 10) // 第一种判断方式  
if (10 == n) // 第二种判断方式
```

【分析】 第二种判断方式更好，因为如果是手误导致代码中少了一个“=”，那么编译时编译器会报错（不能使用赋值操作符“=”对常量进行赋值），帮我们检测出这个错误，减少了代码出错的可能性。

题 30 请说说你是如何使用 const 关键字的？

【分析】 要是在考官面前仅仅回答“const 意味着常数”，考官的脸上一定会晴转多云。关于 const，在本书中也讨论了很多，就其本质，可以认为 const 是一种只读保护。因为它的这个特性，它可以应用在很多场合，起到保护数据的作用。const 的使用形式很多，例如：

```
const int a;  
int const a;  
const int *a;  
int * const a;  
int const * a const;
```

这也是考官常常出的一道让我们辨析 const 的各种使用形式的题目。现在大家应该已经知道，前两个 const 的作用是一样的，表示 a 是一个常整型数。第三个 const 意味着 a 是一个指向常整型数的指针，也就是说，该整型数是不可修改的，但指针可以修改。第四个 const 表示 a 是一个指向整型数的常指针，指针指向的整型数是可以修改的，但指针是不可修改的。最后一个 const 意味着 a 是一个指向常整型数的常指针，指针指向的整型数是不可修改的，同时指

针也是不可修改的。

如果能够正确回答这些问题，一定会让考官脸上多云转晴。如果还能够继续回答使用 `const` 的意义，那么考官脸上一定会阳光灿烂。我们为什么如此钟爱 `const`？

- 关键字 `const` 的作用是给读代码的人传达非常有用的信息，实际上，声明一个参数为常量是想告诉用户这个参数的应用目的。如果你曾花很多时间清理其他人留下的垃圾代码，那么很快就会感谢这点额外的信息，它会告诉你哪些代码应该保留，哪些代码又该进行整理。当然，懂得用 `const` 的程序员很少会留下垃圾代码来让别人清理。
- `const` 关键字可以给优化器一些附加的信息，使用关键字 `const` 也许能产生更紧凑的代码。
- 合理地使用关键字 `const` 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简言之，这样可以减少 bug 的出现。

题 31 同样是表示浮点数，`float` 与 `double` 该如何选择？

【分析】 作为一门应用广泛的程序设计语言，C++ 提供了丰富的数据类型以满足不同的需要。仅为了表示浮点数，C++ 语言就提供了 `float` 及 `double` 等数据类型。我们知道，不同的数据类型所能够表达的数值不同，所占用的内存资源也不同。在开发实践中，有经验的程序员往往会根据实际情况选择恰当的数据类型，这样不仅可以满足表达数值的需要，也可以避免内存资源的浪费。所以，能否选择恰当的数据类型成为一个程序员是否有经验的标志。

`float` 和 `double` 是用来表示浮点数的常见的两种数据类型，那么在这两者之间又该如何选择呢？

针对这个问题，有两种不同的答案。

简单的答案就是，如果程序的精度不太重要，则可以考虑使用 `float` 数据类型；反之，如果程序的精度非常重要，比如要计算卫星的轨道等高精度的问题，则应该选择 `double` 数据类型。

复杂一点的答案就需要了解 `float` 和 `double` 背后的区别了。浮点数主要应用于对精度要求比较高的数学或科学计算应用程序。对于 `float`，它能够保证小数点后 6 位的正确性，而 `double` 的精度更高一些，它可以达到小数点后 15 位。

对于商务软件来说，使用 `float` 或者 `double` 表示浮点数并不是一个好的选择，有些国家甚至禁止在商务软件中使用这两种数据类型来表示浮点数。作为替代，它们使用 10 的幂运算来表示浮点数。例如，它们将 1.23 表示为 123×10^{-2} ，这样整个数字中就不存在浮点数，也就没

有任何精度的损失了。在 C++ 中，也可以很轻松地创建一个以这种无精度损失的方式处理浮点数的类，它可以避免很多因为精度损失而引起的奇奇怪怪的问题。

题 32 以下两种编码风格，哪一种更好？

第一种：

```
// 获取姓名
void GetName(Human* pHuman)
{
    // 判断参数 pHuman 是否有效
    if(nullptr != pHuman)
        return pHuman->GetName();

    return "";    // 返回默认值
}
```

第二种：

```
// 获取姓名
void get_name(human *phuman) {
    if(phuman != nullptr)
        return phuman->get_name();
    return "";
}
```

【分析】 从表面上看，虽然这是一道简单的关于编码风格的选择题，但是却暗藏杀机，稍不注意就有可能掉入出题人设置的陷阱。出题人希望看到的答案，不是“第一种更好”，也不是“第二种更好”，而是考查对编码规范的正确认识和理解。

所谓编码规范，是在项目进行过程中所制定的关于编码格式、注释风格的书写规范，它可以极大地提高代码的可读性，增加代码的可维护性。但是，世界上并没有一种所谓的“最好的”编码规范，即使是现在所流行的各种编码规范也都各有其优缺点，没有普遍适用的标准。如果我们所在的项目团队已经有了一份编码规范，那么就可以按照上面说的做。如果硬要推翻重来，那么可能会带来更多的争吵而不是把问题解决。从商业角度来看，只有两件事是重要的：一是代码可读性好，二是团队中的每个成员都使用相同风格。

因此，我们不要妄图去制定一种“最好的”编码规范，只能结合自己的项目实际，同时参照现在流行的编码规范，采纳其优点，摒弃其缺点，制定出一种“最适合”的编码规范，并且在项目实践中认真严格地执行，这就是“最好的”编码规范。

13.3 考查智力

任何公司都希望招聘聪明的人为它工作，特别是一些国外的大公司，喜欢考一些稀奇古

怪的问题。实际上，这类问题不一定有所谓的标准答案，考官们要看你的创意、你的想法。所以，面对这类问题，答案的正确性不是最重要的，你的创意和想法才是最重要的。

题 33 求下面这个函数的返回值（微软笔试题）。

```
int func(x)
{
    int countx = 0;
    while(x)
    {
        countx++;
        x = x & ( x-1 );
    }

    return countx;
}
假定 x = 9 999。
```

【分析】 函数的返回值是 8。这道题考查的不是对函数和运算符的理解，任何敢参加微软笔试的人都很清楚这个函数的运算过程，可惜我们的大脑不是电脑，无法执行这么多循环运算。这道题考查的是我们分析问题的能力，只有抓住问题的本质，才能快速解决问题。在这个问题中，如果将 x 转化为二进制，则可以清楚地看到含 1 的个数，这也就是我们需要的答案。

题 34 写一个函数找出整数数组中第二大的数（微软笔试题）。

【分析】 通常我们都是写一个函数寻找数组中最大的数，可惜微软要的不是会写寻找最大数的人，它要的是会寻找第二大数的人。你是不是它们要找的人呢？下面给出的代码只是一个参考，也许你还有更好的方法。如果有，不妨发一封邮件给微软，也许一个工作机会就到手了。

```
// 定义最小的整数
const int MINNUMBER = -32767;
// 寻找数组中第二大的数
int find_sec_max( int data[] , int count)
{
    // 假定数组中的第一个数为最大的数
    int maxnumber = data[0] ;
    // 假定第二大数为最小的整数
    int sec_max = MINNUMBER ;
    // 循环遍历数组
    for ( int i = 1 ; i < count ; i++)
    {
        // 如果当前数大于最大数
        if ( data[i] > maxnumber )
        {
            // 原先的最大数成为第二大数
            sec_max = maxnumber;

```

```

        // 当前数成为最大数
        maxnumber = data[i] ;
    }
    else // 当前数小于最大数
    {
        // 继续判断当前数是否大于第二大数
        if ( data[i] > sec_max )
            sec_max = data[i] ;    // 如果大于, 则当前数成为第二大数
    }
}

// 返回第二大数
return sec_max ;

```

题 35 丢失的 1 块钱到哪里去了 (Google 笔试题) ?

3 个人一起住酒店, 每个人出 10 块钱, 一共 30 块钱, 经理说今天优惠, 只要 25 块钱, 叫服务生退 5 块钱给他们, 服务生从中藏了 2 块钱进了自己的腰包, 找回每人 1 块钱, 那么, $3 \times (10 - 1) + 2 = 29$, 丢失的那 1 块钱到哪里去了呢?

【分析】 这道题考查的是逻辑思维能力。大家不要被题目表面上正确的计算公式吓倒, 实际上, $3 \times (10 - 1)$ 是乘客付出的钱, 而 2 是服务员扣的钱, 两者根本是两种性质的钱, 一个是付出, 一个是收入, 根本不能相加在一起。题目中这么计算, 是在偷换概念。正确的计算公式应该是:

$$3 \times (10 - 1) = 27 = 25 + 2$$

这样一计算, 1 块钱都没有少。



接下来该读什么书

当你翻到这一页的时候,就表示你即将完成本书的学习。通过本书的学习——浏览了 C++ 世界的绮丽风光。我想大家已经是初入 C++ 世界的大门了。那么,下一步该做什么呢?我们现在只是初入 C++ 世界的大门,对 C++ 世界有了一个大致的初步了解,俗话说,无限风光在险峰, C++ 世界还有很多高峰等着我们去攀登。要想攀登这些高峰,必须借助更多的书本,用这些书本作为我们攀登高峰的阶梯。

学习国学要熟读四书五经,在 C++ 世界中,也同样有四书五经。这四书五经都是 C++ 世界的经典著作,它们就像武林中的武功秘籍一样,熟读这些经典,我们的 C++ 功力自然可以大增,从而攀登更高的山峰,欣赏更加绮丽的 C++ 世界风光。

A.1 开山鼻祖:《C++程序设计语言》

《C++程序设计语言》可以说是 C++ 世界的开山鼻祖,不管怎样赞誉这本书都不过分。这本书是在 C++ 语言和程序设计领域具有深远影响、畅销不衰的著作,由 C++ 语言的设计者编写,对 C++ 语言进行了最全面、最权威的论述,覆盖标准 C++ 语言及由 C++ 语言所支持的关键性编程技术和设计技术。本书英文原版一经面世,即引起业内人士的高度评价和热烈欢迎,先后被翻译成德、希、匈、西、荷、法、日、俄、中、韩等近 20 种文字,数以百万计的程序员从中获益,是无可取代的 C++ 经典力作。

在本书英文原版面世 10 年后的今天,特别奉上 10 周年中文纪念版(见图 A-1),希望众多具有丰富实践经验的 C++ 程序员能够温故而知新,印证学习心得,了解更加本质的 C++ 知识,让获得的理论运用得更加灵活,也期望新的 C++ 程序员认识到这本书的价值所在,从更高的起点出发,书写更加精彩的程序设计人生。



图 A-1 《C++程序设计语言》

作者简介：Bjarne Stroustrup，英国剑桥大学计算机科学博士，C++语言的设计者和最初的实现者，也是《C++程序设计原理与实践》和《C++语言的设计和演化》的作者。他现在是德州农工大学计算机科学首席教授，同时还是 AT&T 贝尔实验室特别成员。1993 年，由于在 C++ 领域的重大贡献，他获得 ACM 的 Grace Murray Hopper 大奖并成为 ACM 院士；2008 年，他又获得 Dr. Dobb's 杂志的程序设计杰出奖。在进入学术界之前，他在 AT&T 贝尔实验室工作。他是 ISO C++ 标准委员会的创始人之一。

A.2 初学者必看：《C++ Primer 中文版（第 4 版）》

《C++ Primer 中文版（第 4 版）》（见图 A-2）是久负盛名的 C++ 经典教程，其内容是 C++ 大师 Stanley B. Lippman 丰富的实践经验和 C++ 标准委员会原负责人 Josée Lajoie 对 C++ 标准深入理解的完美结合，已经帮助全球无数程序员学会了 C++。本版本对前一版本进行了彻底的修订，内容经过了重新组织，更加入了 C++ 先驱 Barbara E. Moo 在 C++ 教学方面的真知灼见，既显著提高了可读性，又充分体现了 C++ 语言的最新进展和当前业界的最佳实践。书中不但新增大量教学辅助内容用于强调重要的知识点、提醒常见的错误、推荐优秀的编程实践、给出使用提示，还包含大量来自实践的示例和习题。

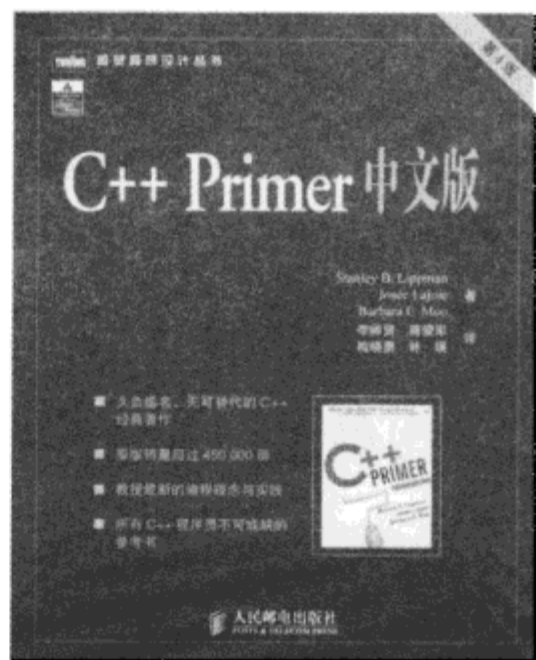


图 A-2 《C++ Primer 中文版（第 4 版）》

对 C++ 基本概念和技术全面而且权威的阐述、对现代 C++ 编程风格的强调，使这本书成为 C++ 初学者的最佳指南；对于中高级程序员，这本书也是不可或缺的参考书。

作者简介：Stanley B. Lippman 的职业是提供关于 C++ 和面向对象的训练、咨询、设计和指导。他在成为一名独立咨询顾问之前，曾经是迪士尼动画公司的首席软件设计师。当他在 AT&T 贝尔实验室的时候，领导了 cfront 3.0 版本和 2.1 版本的编译器开发组。他也是 Bjarne Stroustrup 领导的 AT&T 贝尔实验室 Foundation 项目的成员之一，负责 C++ 程序设计环境中的对象模型部分。他还撰写了许多关于 C++ 的文章。目前他已受雇于微软公司，负责 Visual C++ 项目。

A.3 百科全书：《代码大全，第 2 版》

《代码大全，第 2 版》（见图 A-3）是著名 IT 畅销书作者、《IEEE Software》杂志前主

编、具有 20 年编程与项目管理经验的 Steve McConnell 10 年前的经典著作的全新演绎:第 2 版做了全面的更新,增加了很多与时俱进的内容,包括对新语言、新的开发过程与方法论的讨论,等等。这是一本百科全书式的软件构建手册,涵盖了软件构建活动的方方面面,尤其强调提高软件质量的种种实践方法。

作者特别注重源代码的可读性,详细讨论了类和函数命名、变量命名、数据类型和控制结构、代码布局等编程的最基本要素,也讨论了防御式编程、表驱动法、协同构建、开发者测试、性能优化等有效开发实践,这些都服务于软件的首要技术使命:管理复杂度。为了培养程序员编写高质量代码的习惯,书中展

示了大量的高质量代码示例(及用于对比的低质量代码),提高软件质量是降低开发成本的重要途径。除此之外,本书归纳总结了来自专家的经验、业界研究以及学术成果,列举了大量软件开发领域的真实案例与统计数据,提高了本书的说服力。

这本书中所论述的技术不仅填补了初级与高级编程实践之间的空白,而且也为程序员们提供了一个有关软件开发技术的信息来源。本书对经验丰富的程序员、技术带头人、自学的程序员及没有太多编程经验的学生都是大有裨益的。可以说,只要您具有一定的编程基础,想成为一名优秀的程序员,阅读本书都不会让您失望。

作者简介: Steve McConnell 是 Construx Software 公司首席软件工程师。他是软件工程知识体系(SWEBOK)项目构建知识领域的先驱。Steve McConnell 是以下著作的作者:《快速软件开发》(《Rapid Development》, 1996)、《软件项目生存指南》(《Software Project Survival Guide》, 1998)和《专业软件开发》(《Professional Software Development》, 2004)。他的书作为杰出软件开发的书籍,曾两次获得《Software Development》杂志的震撼大奖。1998 年, Steve McConnell 被《Software Development》杂志的读者评为软件业最具影响力的三大人物之一,与比尔·盖茨(Bill Gates)和李纳斯·托瓦兹(Linus Torvalds)齐名。Steve McConnell 还是 SPC (Software Productivity Center, 加拿大软件进程改进公司)的 ESTIMATE Professional (一款计划和估算工具)的主要开发者, Software Development Productivity Award (软件开发生产力大奖)的获得者。Steve McConnell 从 1984 年就开始从事桌面软件产业,现在在快速开发方法论、工程估算、软件架构实施、性能调整、系统整合和第三方合同管理方面已经具有专业的技术水平。

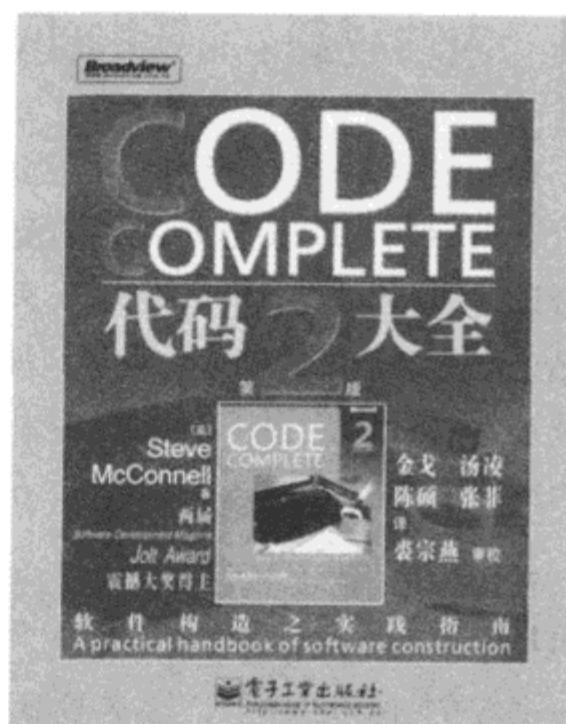


图 A-3 《代码大全(第2版)》

A.4 内功秘籍：《Effective C++ (Third Edition)》

有人说 C++ 程序员可以分成两类，读过《Effective C++(Third Edition)》（见图 A-4）的和没读过的。世界顶级 C++ 大师 Scott Meyers 成名之作的第三版的确承受得起这样的评价。当您读过这本书之后，就可获得迅速提升自己 C++ 功力的一个契机。

在国际上，这本书所引起的反响波及整个计算机技术出版领域，余音至今未绝。几乎在所有 C++ 书籍的推荐名单上，这本书都会位于前三名。作者高超的技术把握力、独特的视角、诙谐轻松的写作风格、独具匠心的内容组织，都受到了极大的推崇和仿效。这种奇特的现象只能解释为人们对这本书由衷的赞美和推崇。

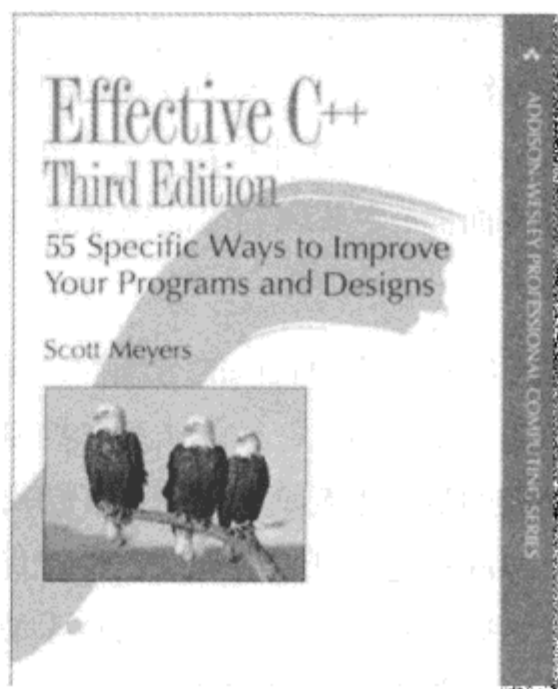


图 A-4 《Effective C++ (Third Edition)》

这本书不是读完一遍就可以束之高阁的快餐读物，也不是用以解决手边问题的参考手册，而是需要您去反复阅读体会的——C++ 是真正程序员的语言，背后有着精深的思想与无与伦比的表达能力，这使得它具有类似宗教般的魅力。希望这本书能够帮助您跨越 C++ 的重重险阻，领略高处才有的壮美风光，做一个成功而快乐的 C++ 程序员。

作者简介：Scott Meyers，世界顶级的 C++ 软件开发技术权威之一。他是畅销书《Effective C++》系列（《Effective C++》、《More Effective C++》和《Effective STL》）的作者，又是创新产品《Effective C++ CD》的设计者和作者，也是 Addison Wesley 的“Effective Software Development Series”顾问编辑，The C++ Source (<http://www.artima.com/cppsource/>) 咨询板块专家，布朗大学计算机科学博士。他的网站是 www.aristeia.com。

A.5 经验很重要：《C++编程规范》

《C++编程规范》（见图 A-5）是一本关于 C++ 编程经验的书。良好的编程规范可以改善软件质量，缩短上市时间，提升团队效率，简化维护工作。在这本书中，两位全世界最受尊敬的 C++ 专家将全球 C++ 社区的集体智慧和经验凝结成一整套编程规范。这些规范可以作为每个开发团队制定实际开发规范的基础，更是每位 C++ 程序员应该遵循的行事准则。这本书实际上涵盖了 C++ 程序设计的各个方面，包括设计和编码风格、函数、操作符、类的设计、继承、构造与析构、赋值、名字空间、模块、模板、泛型、异常、STL 容器和算法等。书中

对每条规范都给出了言简意赅的叙述，并辅以实例说明；书中还给出了从类型定义到错误处理等方面的大量 C++ 最佳实践，包括许多最新总结和标准化的技术，即使使用 C++ 多年的程序员也会从中获益很多。这本书适合于各层次的 C++ 程序员，也可作为高等院校 C++ 课程的教学参考书。

作者简介：Herb Sutter 是 ISO C++ 标准委员会的主席，《C/C++ Users Journal》杂志特邀编辑和专栏作家。他目前在微软公司领导 .NET 环境下 C++ 语言扩展的设计工作。除这本书外，他还撰写了三本广受赞誉的图书：《Exceptional C++ Style》、《Exceptional C++》和《More Exceptional C++》。Andrei Alexandrescu 是世界顶尖的 C++ 专家，《C++ Users Journal》杂志的专栏作家，他的《Modern C++ Design》一书曾荣获 2001 年最佳 C++ 图书称号。书中所开发的 Loki 已经成为最负盛名的 C++ 程序库之一。

最后，我想说的是，C++ 是用来练的，不是用来看的。所以，现在就放下你手中的这本书，开始启动 Visual Studio 编写 C++ 代码，实现你的程序梦想吧！

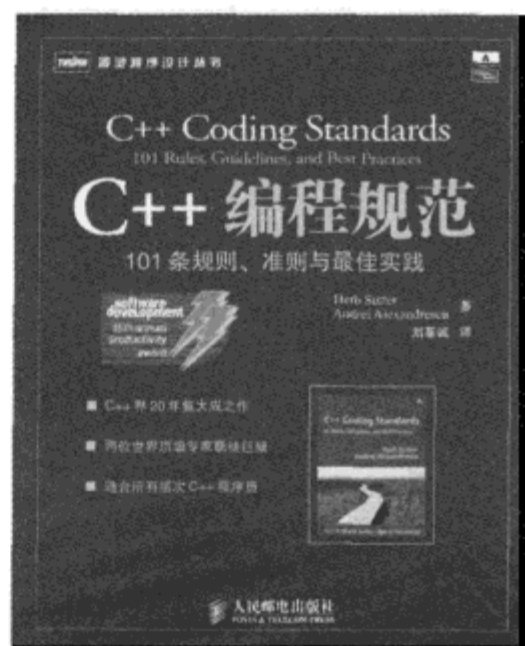


图 A-5 《C++编程规范》

程序也娱乐，开发更武侠。

话说“唐僧”师徒 4 人正前往西天取经，一日夕阳西斜，走得乏了，正想找个客栈打尖休息。还是“悟空”眼尖，远远看见前面有间客栈：“师傅，你看，前面有间客栈。”悟空喊道。

唐僧正骑在白龙马上打瞌睡，听悟空这么一喊，顿时来了精神，抬眼望去，果然看见前面不远处有间客栈，便招呼徒弟们：“悟空、悟能、悟净，还有白龙马，我们今日就在这里休息吧！”（真够啰唆的）说完拍拍马屁股，径直朝客栈前去。白龙马屁股上被八戒烙上去的“BMW”三个大字在夕阳的余晖照射下，甚是醒目。

不一会，唐僧师徒 4 人便到了客栈。小二牵了白龙马到后院去，4 人来到客栈大厅，找个安静的角落入座了，准备要些饭菜吃了好休息，明日趁早赶路。刚入座，悟空使用火眼金睛将整个大厅扫了一番，看看有没有隐藏的“妖怪”。只见大厅七八张桌子，十几个食客，甚是平常。只是堂上多了个说书的，一人，一桌，一椅，一鼠标，再加上个扩音喇叭，在那儿说着什么乔峰、盖聂、燕小六、“瘟到死七”，不知道是哪朝哪代的成年旧事。悟空见这人行有些怪异，恐是妖怪，便拦住店小二打听：“小二，这人是谁啊？在那聒噪些什么乱七八糟的？”

“客官你说那个说书的啊，据说是个程序员，还是×××（某著名品牌）最有价值的专家。这不，最近金融危机，下岗再就业啦，仗着小时候看过几本武侠小说，死皮赖脸地在这儿摆摊说书了，赶都赶不走，老板见没有坏处，也就由他去了。”小二说完，拿眼瞟了瞟正唾沫横飞的说书人，自己忙去了。

悟空听小二这么一说，也就放下心来，转身对唐僧说：“师傅别担心，只是个说书的。专家？想我东土大唐，文昌武盛，专家论斤卖！”

不料那说书的耳朵倒挺灵，悟空这几句鄙夷的话，悉数被他听了去，不由得恼羞成怒。于是把桌子上的鼠标一拍，对着悟空正色说道：“这位客官可就误会了，其实我是一个程序员！Programmer, understand?”

说完不再理会悟空的反应，自顾自地说了下去：“我想各位听众江湖恩仇也听烦了，不

如我来给大家说说软件开发界的奇闻轶事，甚是有趣。”

众食客早就不耐烦了他那几个说了几百遍的段子，都能倒背如流了。听他要改变话题，都纷纷来了兴致，鼓掌叫好，看他要说个什么新鲜名堂。

说书的像是受到了鼓舞，把鼠标重重地一拍，说出下面这样一段书来。

话说这 21 世纪，是软件的世纪。至于软件的重要性，自然不需要多说。今个就从软件的起源地说起。软件本起源于美国，后才传入我国。所以说功力深厚，还数人家美国。美国有两个闻名的软件帮派，一曰 CodeGuru，一曰 CodeProject。这两帮派高手如云，一如我中土的武当少林。CodeGuru 先不说，单说这 CodeProject。CodeProject 创立于 1999 年，至今已有 10 余年的历史了。CodeProject 是一个免费公开源代码的程序设计网站，使用者主要是 Windows 平台上的电脑程序设计人员。该网站上的每篇文章几乎都附有源码（src）和例子（demo）供大家下载。经过 10 余年的发展，现在已家喻户晓，成为每个 Windows 程序员必收藏的网站。

CodeProject 的“掌门人”根据近期的热点，每周出一道调查题，各位访客根据自己的情况，用鼠标点点找到适合自己的答案，此之谓“用鼠标说话”。这不，情人节已近，“掌门人”出了这样一道题，如图 1 所示。问各位程序员对自己的情人（也就是自己使用的开发语言）钟情几许？

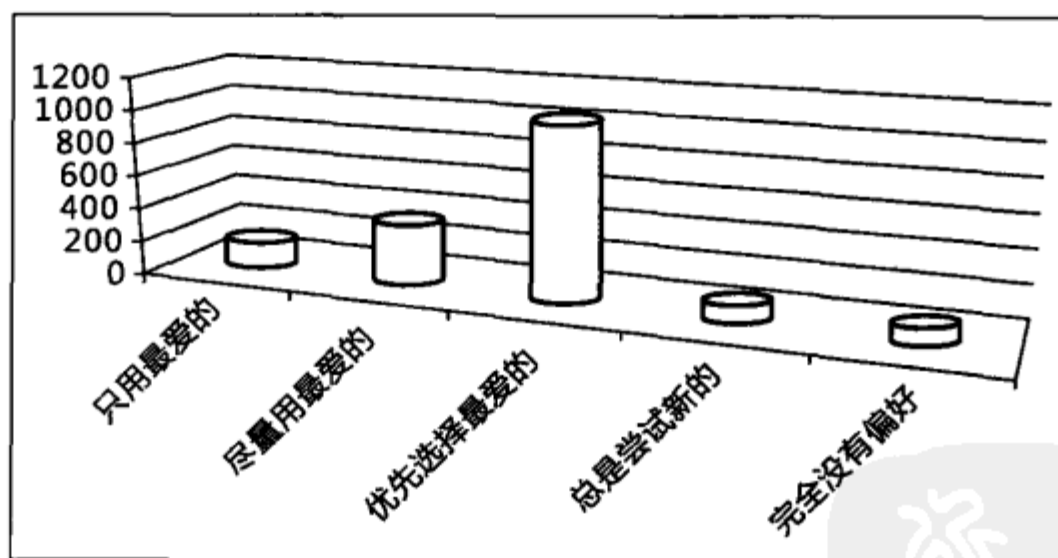


图 1 “你到底爱我有多深”大调查

调查结果甚是耐人寻味。

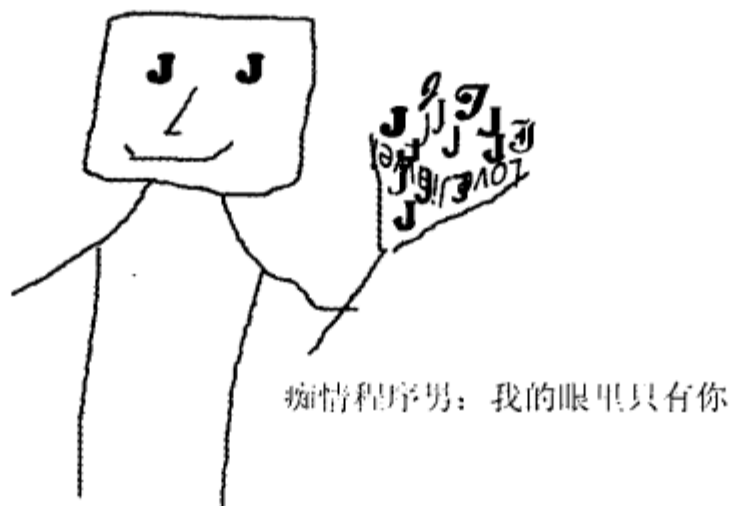
超过 60% 的程序员这样说：“Darling，我爱你，但是有时候我要和别人在一起！”

这部分程序员虽然有自己喜欢的语言，但是不得不使用其他语言工作。

超过 20% 的程序员总是尽量使用自己最喜欢的语言。比起前一种程序员，这种程序员算是比较钟情的了。

最后不到 20% 的程序员，大声地说出了“我的眼里只有你”！如图 2 所示。

这些程序员只自己喜欢的语言工作，只用自己钟情的语言表达自己的思想。这种人，简直是程序员中的杨过，对自己喜欢的开发语言情有独钟。女生找男朋友，就应该找这样的！



容，凭借接近 C 语言的效率，在工业界占据了相当多的份额。在以后的发展中，C++语言不断引入新的内容，比如标准模板库（STL）和后来的 Boost 等程序库的出现、泛型程序设计的流行，使得 C++语言牢牢占据了 TIOBE 编程语言排行榜前三的位置，成为业界最流行的编程语言之一。如果要问程序员们最喜欢什么开发语言，他们大多数都会回答“C++语言”，如图 3 所示。

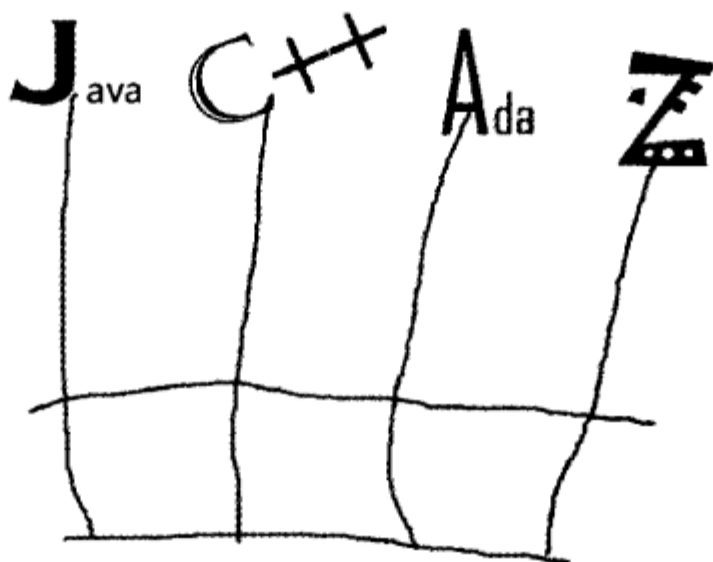


图 3 程序员的十八般兵器

说书人不由得感慨了一番，忽然觉得脑子里有点东西，想写下来。于是拿出自己的笔记本（不用电的那种笔记本啊），写下了下面几段话。

- 60%的程序员是实用主义者，哪管他什么语言，只要能解决问题就行。就像领导人说的，管它黑猫白猫，只要抓到老鼠就是好猫。语言只是工具而已，解决问题才是硬道理。那不到 20%的痴情程序员，是好“老公”的料，却不一定是个好程序员。当然，大师级的人物除外。
- 世间程序设计语言如此之多，没有高低贵贱之分，只有擅长的领域不同。不同语言，自有它各自擅长的不同领域。常常有人问学习某某语言有没有前途，其实不能问语言有没有前途，需要先问你希望从事什么领域。你所要从事的领域，自有它钟情的语言，学习它就好了。
- 学好 C++语言，走遍天下都不怕！
- 那个猪头猪脑的和尚甚是讨厌，下回有机会也说他一说！

说书人写完，合上笔记本，顺手拿起书桌上的那本《我的第一本 C++书》，与他的 C++情人相会去了。



The world is built on C++.

—*Herb Sutter*

*the chair of the ISO C++ standards committee
and chief native languages architect at Microsoft*

看得有趣、学得轻松

看图学 C++

陈良乔

导读：

看图也能学 C++？！

没错，看图也能学 C++！

这本迷你书是《我的第一本 C++ 书》的迷你版，它抽取了《我的第一本 C++ 书》中的全部的精美插图，并配上相应的解释说明。它以图文并茂的生动形式，向你讲解那些所谓的高深的 C++ 知识，让你对那些抽象的 C++ 知识有一个更加形象的理解，向你展示这个美丽而神秘的 C++ 世界，让你在有趣的看图过程中，轻松地学到了 C++ 知识。

看得有趣、学得轻松

更多信息，请访问

<http://imcc.blogbus.com>



第一篇叩开 C++世界的大门



C++世界地图

对一个即将到陌生的地方去旅行的人来说，什么是最重要和必需的？

没错，是一张内容丰富详尽、生动有趣的旅行地图。借助这张地图，我们知道在什么地方停车吃饭、在什么地方打尖住店。即将进入陌生的 C++世界的各位旅行者对 C++世界有太多的问题和疑惑：

C++是什么？

C++是怎么来的？

C++能做什么？

如何学好 C++？

面对这些问题，我们同样需要一张 C++世界的地图。这张 C++世界地图可以为我们解答这些问题和疑惑，让我们清晰地认识 C++世界。同时，我们可以通过这张 C++世界地图，了解 C++世界的整个面貌：有哪些好玩的地方、有哪些有趣的故事、有哪些有用的知识、有哪些危险而需要注意的地方。这张 C++世界地图，将带领我们畅游整个 C++世界。

还等什么，让我们出发吧！



1.2.3 从 C++ 到 .NET Framework 的 CLI

那么，到底什么是 C++/CLI? 它跟传统的 C++ 又有什么不同呢?

CLI 指的是通用语言结构，一种支持动态组件编程模型的多重结构。在整个 CLI 结构中，最重要的是公共语言运行时（Common Language Runtime，CLR），它负责管理微软中间语言（Microsoft Intermediate Language，MSIL）代码的运行环境。CLR 位于 CLI 的下半部分（如图 1-1 所示），主要包括类加载器（Class Loader）、实时编译器（IL To Native Compilers）和一个运行时环境的垃圾收集器（Garbage Collector）。CLI 运行在底层操作系统与程序之间，为 MSIL 代码提供运行的环境，这使得 CLI 成为一个实时的软件层，一个有效的执行系统。我们可以将任何语言编写的代码，通过特定的编译器转换为 MSIL 代码，然后在 CLI 上运行。

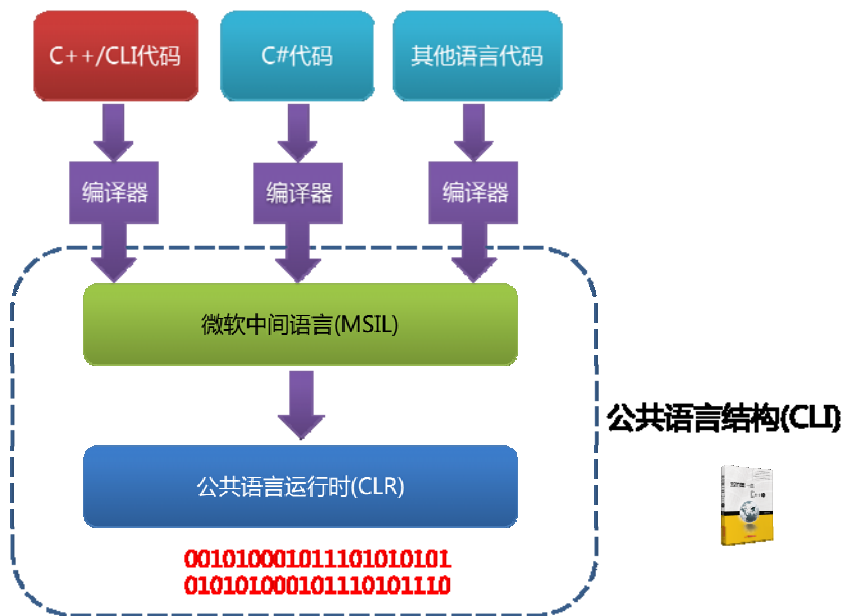


图 1-1 C++/CLI 的结构

1.2.6 五岳剑派：C++世界的五大子语言

C++已经有 40 多年的发展历史了，在发展过程中，因为不同的应用领域，不同的开发思想而形成了不同的 C++子语言。每个子语言各有所长，就像 C++世界的五岳剑派，各自在自己的领域独领风骚，形成 C++世界百花齐放的繁盛局面（如图 1-2 所示）。

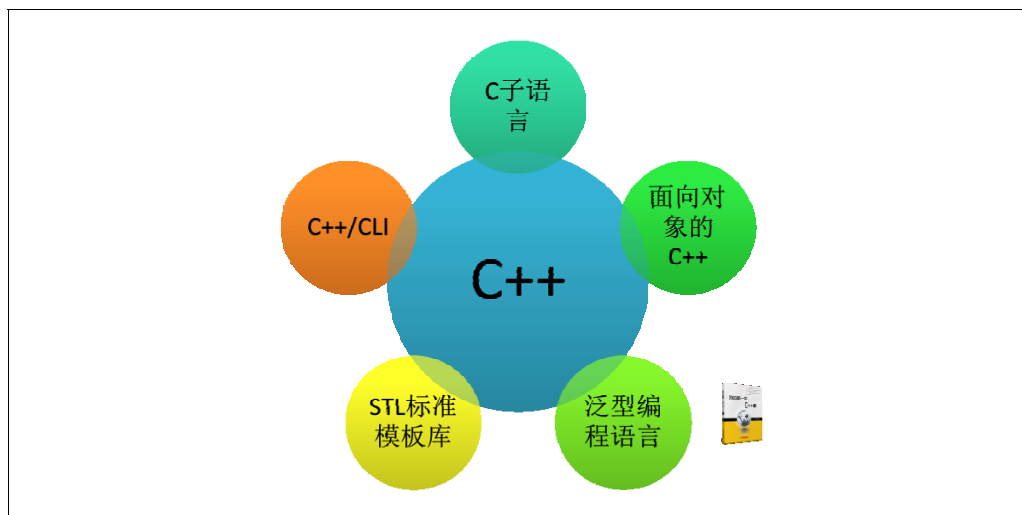


图 1-2 C++的五种子语言争奇斗艳

1.3 三分天下：C++世界版图

C++语言的发展过程，不仅是一个特性不断增加，内容不断丰富过程，更是一个在应用领域不断攻城略地的过程。在其 40 余年的发展过程中，C++在多个应用领域都得到了广泛的应用和发展。无论是在最初的 UNIX 操作系统上，在 Windows 操作系统上，还是在最近兴起的嵌入式系统上，C++都占有一席之地（如图 1-3 所示）。

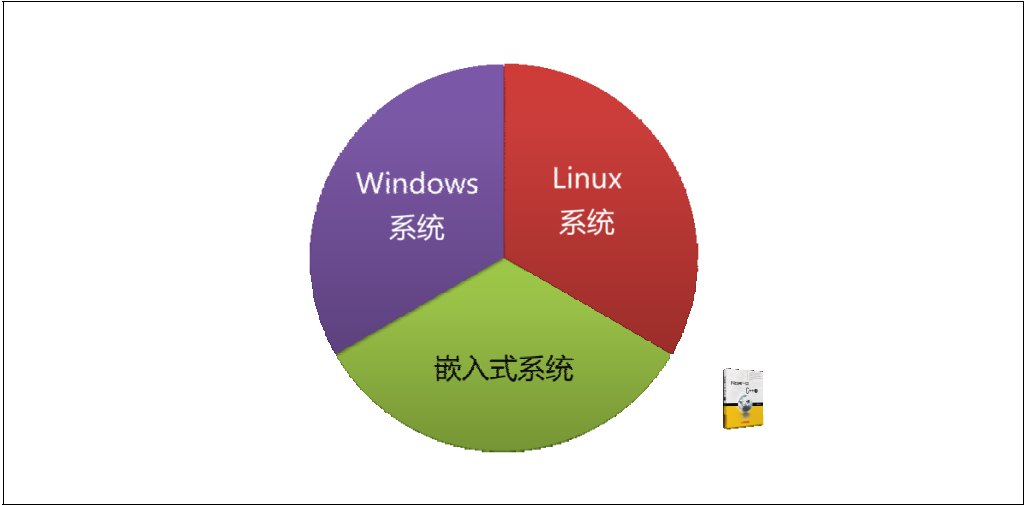


图 1-3 C++世界三分天下

1.4.1 会说话，就会编程：C++是如此简单的编程语言

C++是一门程序设计语言，有着语言的基本特征，我们可以像学习普通语言一样来学习C++。

语言，是用来描述和表达现实世界的，编程语言也不例外。为了描述现实世界的事物，我们需要一些名词，在C++中就是数据类型和用数据类型表达的数据。而为了表达事物之间的关系，将各个事物连缀成句子，在C++中就是表达式。将多个句子通过一定的逻辑关系组合起来，就可以形成一篇文章，同样的，在C++中通过一定的逻辑控制将多个表达式组合起来，就形成了程序。通过C++编程语言和自然语言的对比，我们可以轻松地理解C++程序的含义。C++是描述现实世界的编程语言，编写程序的过程，是将自然语言翻译成程序语言的过程，如此而已。

比如，在自然语言中，我们可以这样来描述一件事情：

有个男孩叫小张，有个女孩叫小芳。男孩向女孩示爱。女孩对男孩进行考察，如果男孩有房又有车，则与之交往；如果没有，则与之拜拜。翻译过程可以参考图 1-4。

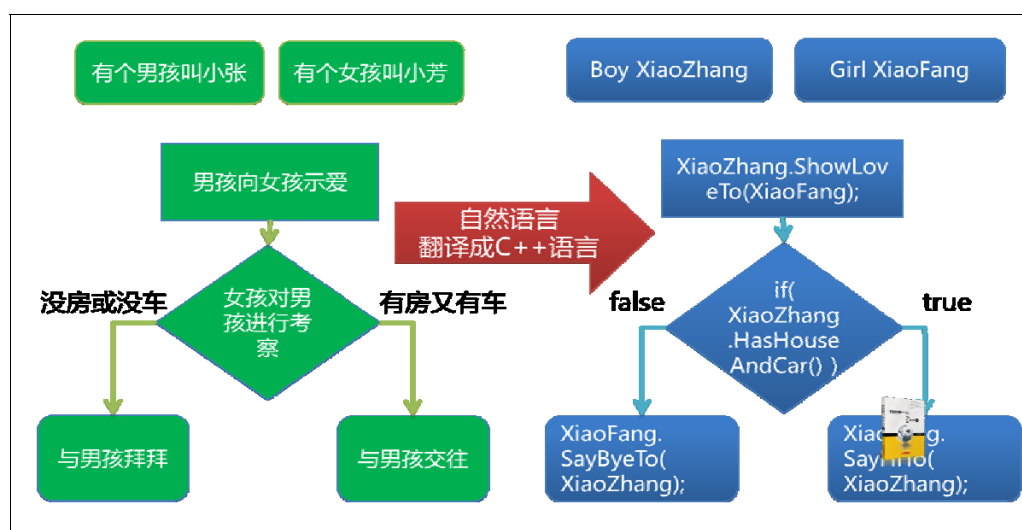
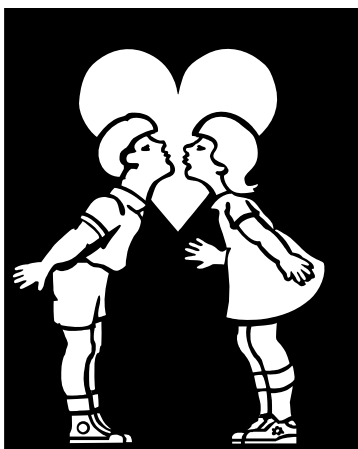


图 1-4 编程就是把自然语言翻译成 C++语言

与 C++ 第一次亲密接触

在浏览了 C++ “三分天下”的世界版图之后，我们对 C++ 有了基本的了解，算是跨入了 C++ 世界的大门。那么，我们下一步该往哪里去呢？即刻开始编写 C++ 程序？还是……？

正在我们犹豫的时候，看到前面有一个人被一群满头问号的 C++ 初学者围在当中。我们赶紧挤进去一看，噢，原来是一个 C++ 程序正做自我介绍呢——



2.1.3 我的五官和四肢：C++程序=预编译指令+程序代码+注释

麻雀虽小，五脏俱全。大家别看我个头小，只有短短的几行代码，实现的功能也很简单，但是我同样拥有健全 C++ 程序的五官和四肢：预编译指令、程序代码和注释，如图 2-5 所示。大多数情况下，这三个基本部分都被放在一个扩展名为“cpp”的文本文件中，这个文件被称为 C++ 源文件。源文件记录了我的五官和四肢、规划了我的人生。而你作为源文件的编写者，就是我的设计师了。通过修改源文件，你可以改变我的面貌、我的人生轨迹，让我完成各种任务。

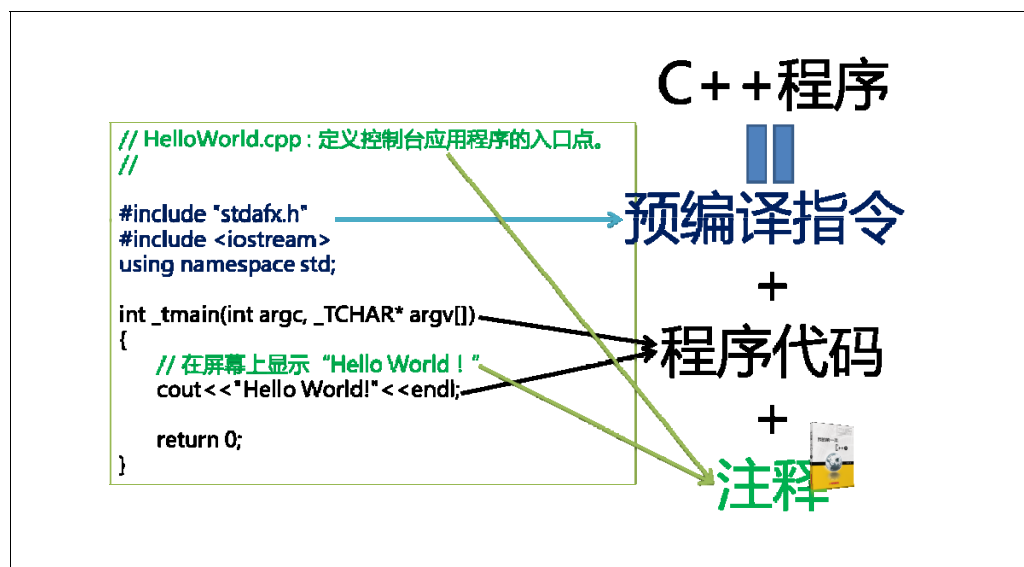


图 2-5 C++程序=预编译指令+程序代码+注释

2.1.4 我的父亲母亲：编译器和链接器

虽然我是 Visual Studio 创建的，但是实际上，我的老爸和老妈是 Visual Studio 集成的编译器和链接器。此外，Visual Studio 提供的主要是编辑功能，让你更方便地编辑我的源代码。

我老爸编译器的工作是将高级语言 C++ 翻译为低级语言（机器语言）。

我的源文件是使用 C++ 这种高级程序设计语言编写的，便于人们编写、阅读和维护。但计算机不理解高级语言，所以老爸的职责是将源程序翻译成计算机能够解读运行的目标语言（target language）。目标语言通常是汇编语言或目标机器的目标代码（object code），有时也称作机器代码（machine code）。通过老爸的工作，计算机能看懂 C++ 程序，就可以按照源文件中的指令执行相应的动作。

老爸完成我的编译工作后，我还只是一些目标文件，还需要老妈链接器将一个或多个由老爸编译生成的目标文件和库函数链接成可执行文件，这样才诞生了一个可执行的 C++ 程序。再来回顾一下我的诞生过程（如图 2-6 所示）：

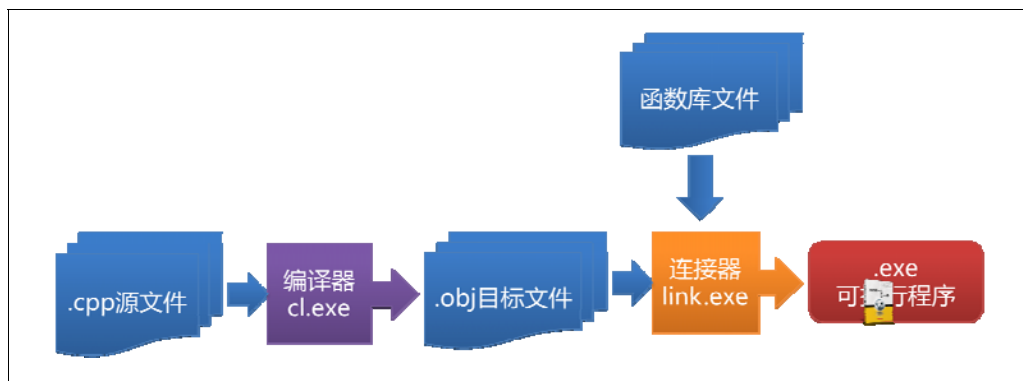


图 2-6 我的父亲母亲

2.1.5 我的一生是这样度过的：C++程序的执行过程

进入_tmain()函数之前的事情我做不了主，但是进入_tmain 函数后，就是我的地盘了。我会按照你在源代码中给我制定的人生规划，一条语句一条语句地往下执行，一步一步地往下走。你一定还记得，我的源代码是这样的：

```
int _tmain(int argc, _TCHAR* argv[])
{
    // 在屏幕上显示 "Hello World!"
    cout<<"Hello World!"<<endl;

    return 0;
}
```

_tmain()函数有两个参数 argc 和 argv，你可以通过这两个参数，给_tmain()函数传递一些信息，给我一些额外的吩咐，比如通过这两个参数告诉我应当在屏幕上显示什么内容等等。在这里，暂时没有使用这两个参数。

进入_tmain()函数后，我遇到的第一个语句就是：

```
cout<<"Hello World!"<<endl;
```

这条语句让我在 DOS 窗口中显示 “Hello World!” 这样一个字符串，于是我开始控制 DOS 窗口，在其中显示这个字符串，完成了你交给我的任务。

接下来的一个语句是：

```
return 0;
```

这条简短的语句宣告了我人生历程的结束。它表示整个_tmain()函数的结束。图 2-7 是我短暂而光辉的一生！



图 2-7 HelloWorld 短暂而辉煌的一生

2.1.6 我的人生目的：描述数据与处理数据

每个人都会问自己人生的目的是什么？我的人生目的是什么？人们编写程序的目的，是为了用程序解决现实世界中的问题。人们观察发现，这些问题都是以数据作为输入，然后对这些数据进行处理，最后得到问题的结论。所以，我人生的目的是描述数据并处理数据，最终解决现实世界的问题，如图 2-8 所示。

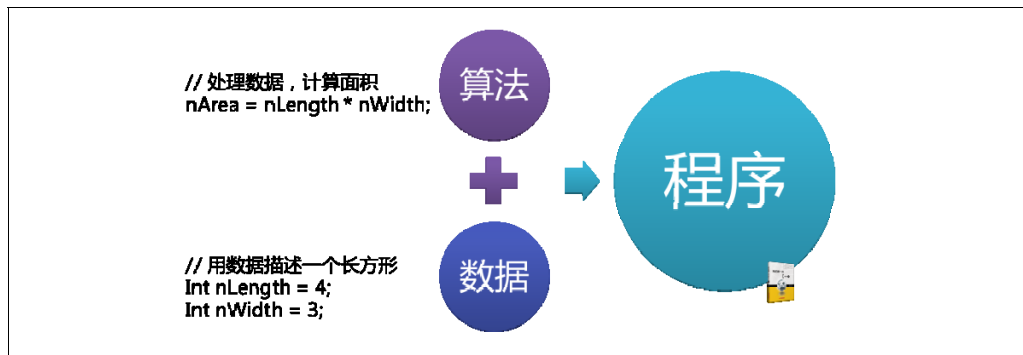


图 2-8 我的人生目的

2.2.3 读写文件

在这段程序中，我们首先创建了一个输入文件流 `ifstream` 的对象 `fin`，并利用它的构造函数将其连接到一个文本文件 `Date.txt`。所谓构造函数，就是这个对象创建的时候所执行的函数。这里，我们使用“`Date.txt`”作为参数来调用这个构造函数，实际上就是使用这个文件创建 `fin` 对象。除此之外，我们还可以使用 `fin` 所提供的 `open()` 函数来打开一个文件。当我们利用 `fin` 成功打开一个文件之后，就可以利用提取符“`>>`”从 `fin` 中提取各种数据。“`>>`”会以空格为分隔符逐个从文件中读取数据并将其保存到相应的数据变量中。例如，如果文件中的内容如下：

```
用户输入的当前日期是：
1983 7 3
```

默认情况下，`fin` 总是从文件的开始部分进行读取的，为了直接读取第二行的内容，我们使用“`fin.ignore(256, '\n');`”忽略了第一行的内容，将读取位置跳转到第二行。然后，通过提取符“`>>`”，我们将第二行用空格分割的三个数据分别提取并保存到了三个变量中。

同样，为了将数据写入文件，我们需要创建一个输出文件流 `ofstream` 的对象 `fout`，然后通过它的构造函数或者是 `open()` 函数来打开一个文件，将这个文件和 `fout` 对象连接起来，然后通过插入符“`<<`”将数据插入到 `fout` 对象，也就实现了将数据写入到它所关联的文件中的目的。整个过程如下图 2-9 所示：

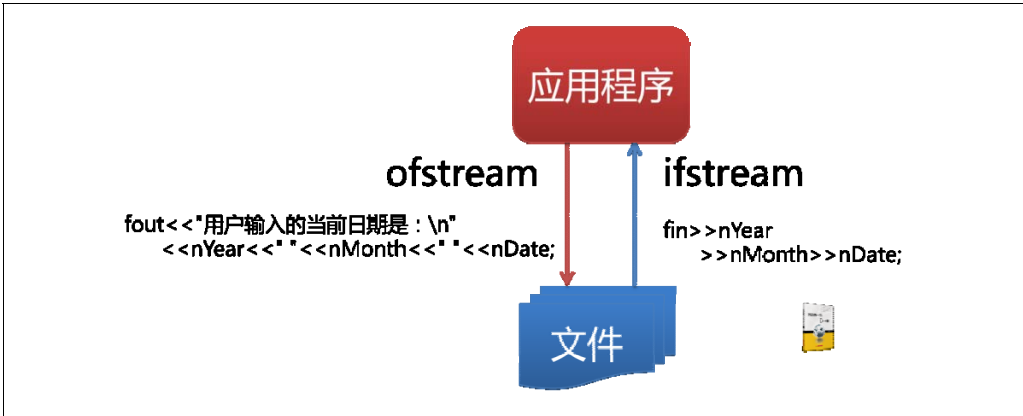


图 2-9 文件读写

第二篇欢迎来到 C++ 世界



· C++世界众生相

在听过了 HelloWorld.exe 的自我介绍，完成了与 C++ 的第一次亲密接触后，大家是不是都急不可待地想要一试身手，开始编写 C++ 程序了呢？

我们知道，程序首先是使用数据来描述现实世界的。当我们尝试使用数据来描述现实世界时，马上就遇到了一个问题：C++ 世界中的数据这么多，我们一个都不认识，该从哪里开始啊？别着急，现在我就来为你们介绍 C++ 世界的芸芸众生：基本数据类型。



3.1 百家姓：C++中的数据类型

我们知道，编程是使用程序设计语言来描述和表达现实世界的。现实世界中有很多客观存在的事物，例如数字、人、车辆等。很多数据是同一类的，比如人的名字，都是由文字构成的；人的身高，都是由数字组成的。在程序设计语言中，我们将这些相同类型的数据抽象成数据结构。数据结构是对现实世界中的同类数据的描述，我们把它也称为数据类型。这就像现实世界中人的姓氏一样，同一个姓氏的人是一家人。在 C++ 世界中，同样数据类型的数据是同一个类别的，也有着相同的一些特征。为了描述现实世界中丰富多样的事物，通常将这些事物定义成具体的数据，而数据类型则是这些事物的种类。数据类型就像 C++ 世界的百家姓，一个数据的数据类型，决定了这个数据是哪一家的人，如图 3-1 所示。

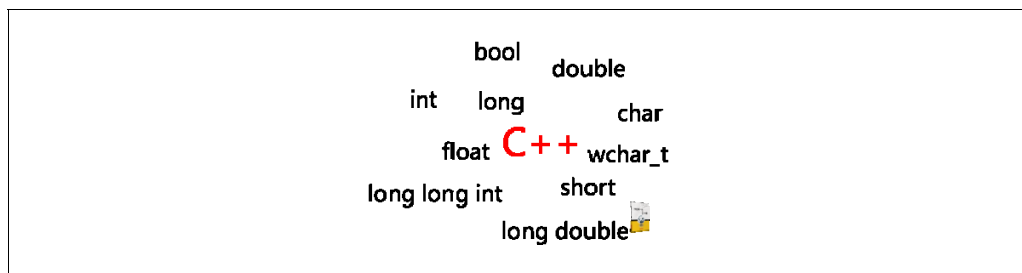


图 3-1 C++的和谐大家庭

3.6.1 排排坐，吃果果：数组的声明与初始化

我们常常遇到这样的数据：数量很大；有相同的数据类型；有相同的处理方式。例如，一个公司所有员工的工资、一个学校所有学生的成绩、一个地区一年的气温，等等。为了描述这种量大且相似的数据，C++提供了数组这种构造型数据类型。

如果把整个内存看成是一座宾馆，那么可以把数组看成是某一层楼上的一个个连续的小房间。这些小房间具有相同的容量，可以存放相同数据类型的数据。当然，房间容量的不同或者连续房间个数的不同，可以在内存中形成不同的数组，如图 3-2 所示。

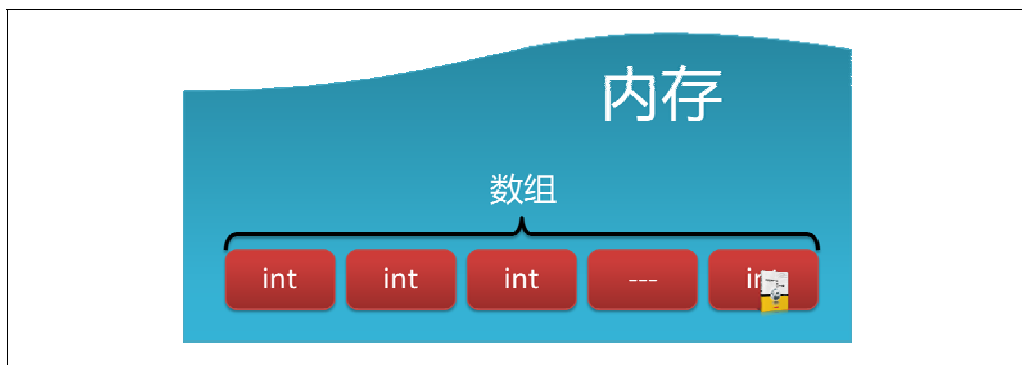


图 3-2 数组就是内存中的多个小房间

3.6.2 数组的使用

定义好数组后，可以引用数组中的数据进行运算。我们想到某个房间找人，需要知道他的房间号。同样，要引用数组中的数据，也需要给定数组的下标，也就是数据在数组中的序号。例如：

```
// 定义一个长度为 100 的整数型数组
int nSalary[100];
// 通过给定数据在数组中的序号
// 读取 nSalary 数组中的第 25 个元素
int n = nSalary[24];
```

其中，nSalary 是定义的数组名，24 表示想要引用的数据是这个数组中的第 25 个数据。这条语句的含义就是把 nSalary 数组中的第 25 个数据赋值给变量 n。大家可能会感到奇怪，为什么要用下标 24 来引用数组中的第 25 个数据呢？这是因为 C++ 数组的下标总是以 0 开始的，也就是说，nSalary[0] 是这个数组中的第 1 个元素，依此类推，下标 1 表示第 2 个元素，下标 24 就表示第 25 个元素了。另外，需要注意的是，下标必须小于数组定义时的大小。简单来讲，一个长度为 n 的数组，其下标最大值是 (n-1)，如图 3-3 所示。例如，nSalary[99] 是合法的，表示这个数组的最后一个元素，而 nSalary[100] 就是非法的。当在程序中使用 nSalary[100] 访问数组时，会访问到数组以外的内存区域，这会产生非常严重的错误。初学者需要谨记这一点，以免一点小问题造成程序崩溃。

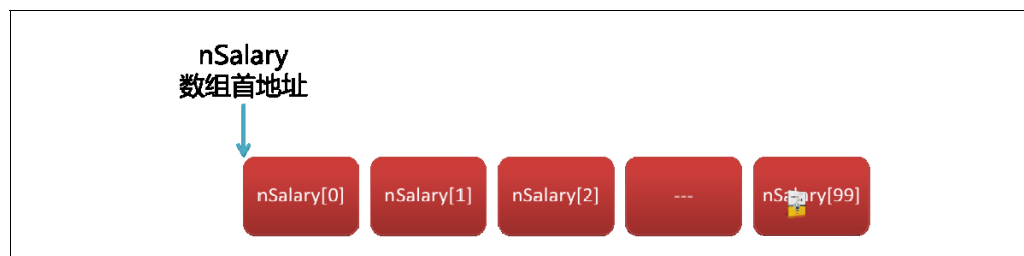


图 3-3 通过下标访问数组

对于二维数组、三维数组等多维数组，同样可以通过给定多个下标来引用数组中的元素。例如：

```
float fA = fArray [1][1];
```

这条语句访问的就是 fArray 多维数组中的第二行第二列的元素。

针对多维数组，C++ 是按照维数从高到低的顺序原则来排列数据的，较高的维数总是得到优先排列，而在同一维，则按照下标从低到高的顺序进行排列。如图 3-4 所示，C++ 在进行二维数组的内存排列的时候，先排列第一维的第一层，这一层包含了第二维的所有数据。在同一层中，第二维再按照下标从低到高的顺序进行排列。完成这一层的排列后，再进行第一维的第二层排列，依此类推。按照这样的方式来理解数组，通过下标来引用数组中的元素

就比较清晰了。

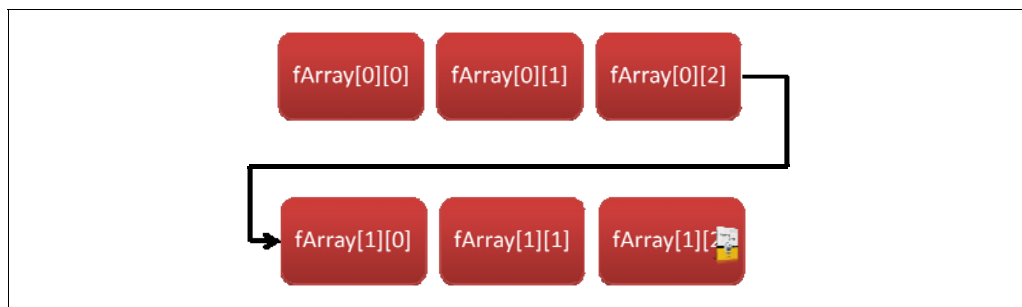


图 3-4 多维数组的存放顺序

通过对数组元素的引用，不仅可以读取它的值，还可以修改它的值。例如：

```
// 对数组中的元素进行赋值  
nSalary[24] = 1200;  
fArray [1][1] = 3.54;
```

通过对数组中元素的引用，可以像使用一个单独的变量一样使用数组中的每个元素。这样在描述大量相似数据的时候，就无须定义多个变量，只需要一个数组就可以了，然后通过不同的下标，就可以访问到不同的数据，就像拥有多个变量一样。要是陈良乔早点学到这一点，他就不会晕倒了，用一个数组，就可解决他的所有问题。

3.8.1 打包复杂：结构体的定义

其中，struct 关键字表示要创建一个结构体，结构体名就是要创建的新结构体的名字，通常使用结构体描述的事物来作为结构体的名字。在结构体的内部，我们分别使用多个不同数据类型的变量来表示复杂事物的各个属性。因为这些变量共同组成了结构体，所以这些变量称为结构体的成员变量。有了结构体，就可以在结构体中定义多个不同类型的成员变量，从各个属性来描述一个复杂的事物。例如，可以这样来定义描述人这个复杂事物的结构体：

```
// 定义结构体 Human 描述人这个复杂事物
struct Human
{
    string    m_strName;           // 姓名
    bool      m_bMale;            // 性别
    int       m_nAge;             // 年龄
    int       m_nHeight;          // 身高
    float     m_fWeight;          // 体重
};
```

以前是用各个基本数据类型的变量来分别描述一个复杂事物的各个属性。这里是将变量集合在一起，打包成一个结构体，如图 3-5 所示。有了结构体，就可以定义一个统一的结构体变量来描述一个具体的复杂事物，代替原来定义多个变量描述同一个事物。例如：

```
// 定义一个 Human 结构体变量描述“陈良乔”这个人
// 这个结构体包含了他的姓名、性别和年龄等信息
Human chenliangqiao;
```

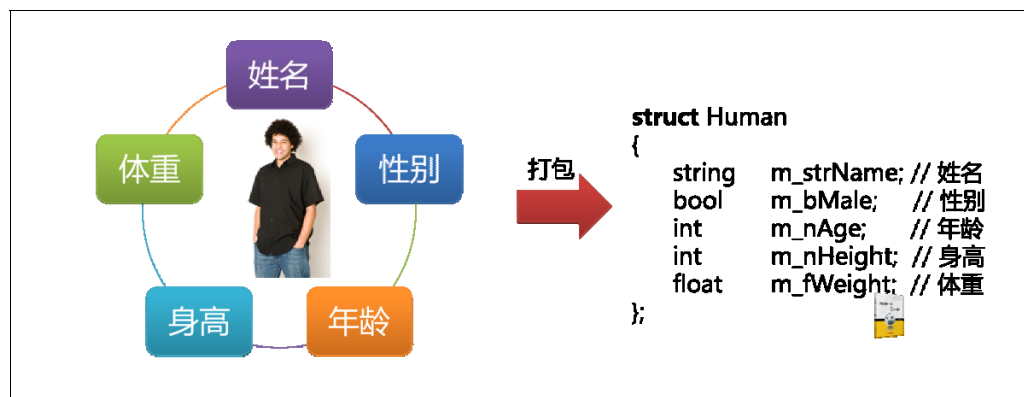


图 3-5 将复杂事物打包成结构体

3.9.1 指针的本质：表示内存地址的数据类型

在典型的 32 位平台上，可以把内存空间看成是由很多个连续的小房间构成的，每个房间

就是一个小存储单元，大小是 1 个字节，房间中住着数据。有的数据比较小，比如一个 char 类型的字符，它只需要一个房间就够了。而有的数据比较大，就需要占用好几个房间，比如一个 int 类型的整数，其大小是 4 个字节，需要 4 个房间才可以安置。为了方便找到住在这些房间中的数据，房间就需要按照一定的规则编号，这个编号，就是通常所说的内存地址。这些编号是用一个 32 位的二进制数来编码的，比如 0x7AE4074B、0xFFFFFFFF 等，如图 3-6 所示。一旦知道某个数据的房间编号，就可以通过这个编号来对相应房间中的数据进行存取操作。C++中为了灵活地操作内存，特别内建了一种特殊的数据类型，以用来存放内存单元的地址，这就是指针。而存放在指针中的内存地址，则可能是一个对象的地址、一个整数的地址，甚至是一个函数的地址。一般来说，如果指针变量所保存的是一个对象或者函数的地址，就说这个指针指向这个对象或者函数。

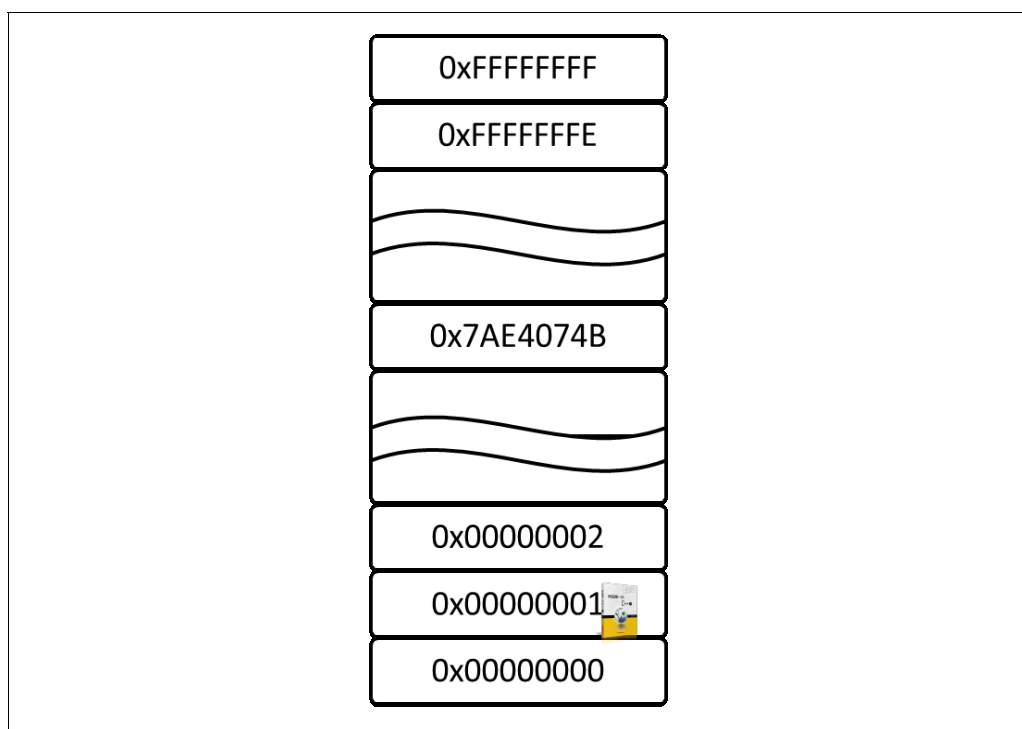


图 3-6 内存被划分为许多小的单元房间

3.9.3 指针的赋值和使用

在得到一个指针变量之后，指针变量的值还是一个随机值。这个值可能是内存中无关紧要的数据，也可能是重要的数据或者程序代码，如果直接使用是很危险的，所以在使用指针之前，必须对其进行赋值，将其指向某个有意义的数据或代码。对指针变量进行赋值的语法格式如下：

```
指针变量 = 内存地址；
```

可以看到，对指针变量的赋值，实际上就是将这个指针指向某一内存地址，而这个内存地址上存放的就是这个指针想要指向的数据。通常我们用一个变量来保存数据，那么该如何方便地得到一个变量在内存中的地址呢？反过来，如果知道一个指针，又如何取出存放在其中的数据呢？为了解决这两个问题，C++提供了两个与内存地址相关的运算符——“&”和“*”。

1. “&”运算符

“&”称为取地址运算符，如果把它放在一个变量的前面，则可以得到该变量在内存中存放的地址。例如：

```
// 定义一个整型变量
int N = 703;
// 取得整型变量的地址并将其赋值给整型指针
int* pN = &N;
```

通过“&”运算符可以取得 N 这个整型变量的内存地址，然后将其赋值给指针 pN，也就是将指针 pN 指向 N 这个整数数据，如图 3-7 所示。

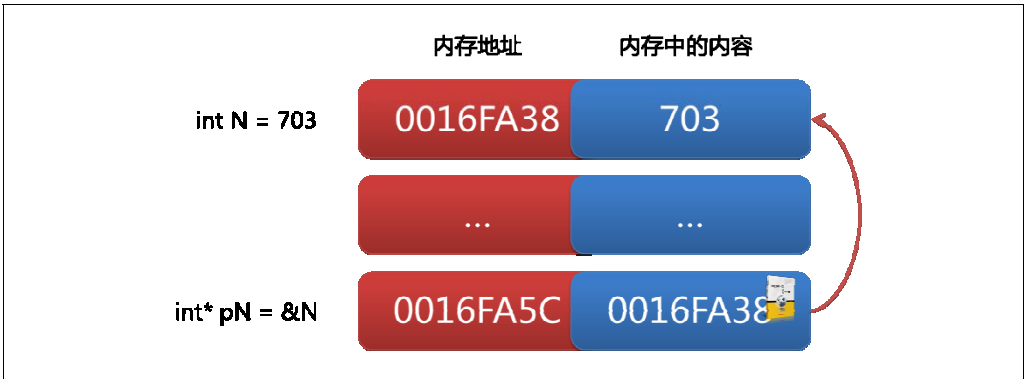


图 3-7 指针和指针所指向的数据

· 串珠成链：将语句编织成程序

见过 C++世界的芸芸众生之后，我们知道如何使用各种类型的数据变量来描述现实世界中的各种事物了。使用数据变量，几乎可以写一个工资统计程序，代码如下：

```
// 工资统计程序
int _tmain(int argc, _TCHAR* argv[])
{
    // 保存所有工资的数组
    int nSalary[100];
    // 保存平均工资、最高工资和最低工资的变量
    float fAverageSalary;
    int nSalaryMax;
    int nSalaryMin;

    // .....对工资进行处理
    return 0;
}
```

在这段代码中，我们用数组保存了所有员工的工资，用各种数据类型的变量保存了平均工资、最高工资和最低工资等，仿佛一个庞大的工资统计程序马上就要实现，大家是不是很有成就感？

抱歉的是，我不得不打破大家的美梦。

我们现在只知道如何用数据描述现实世界，对于如何处理数据以解决问题还一无所知。比如，我们不知道如何用加、减、乘、除计算平均工资；也不知道如何计算最高工资和最低工资。程序的两个目的——描述数据和处理数据，现在只完成了第一步，用变量描述现实世界中的数据；第二步就是要对数据进行处理，最终解决问题。

为了完成工资统计程序，下面来看看在 C++世界中如何对数据进行处理吧！



4.2.1 if 语句

我们总是用“如果……，就……”来表达条件选择，所以，C++也向我们学习，提供了关键字 `if` 来实现选择结构。`if` 语句的语法格式如下：

```
if ( 条件表达式 )  
{  
    语句 1;  
}  
else  
{  
    语句 2;  
}
```

在条件选择语句中，首先计算条件表达式的值，如果表达式的值为 `true`，则执行语句 1；否则执行语句 2。通过使用条件选择语句，可根据条件表达式的不同值而改变程序执行的流程，可以在语句 1 和语句 2 中实现不同的功能。`if` 语句的执行过程如图 4-1 所示。

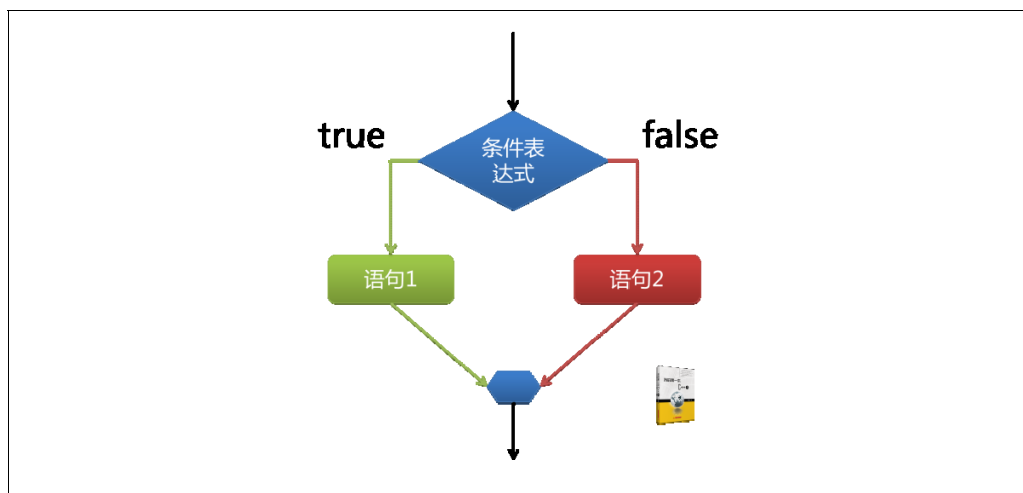


图 4-1 条件选择结构的执行流程

4.2.2 并列的选择：switch 语句

其中，条件表达式就是要进行判断的条件。switch 语句首先计算条件表达式的值，这个表达式的值只能是整型或字符型。完成这个表达式的计算之后，程序开始在各个“case”分支中从上到下逐个匹配，查找哪个常量值和这个表达式的值相等。如果找到相等的常量表达式，则以此为入口开始往下顺序执行 case 分支中的语句，直到遇到 break 关键字，完成整个 switch 语句的执行。如果查找所有 case 分支都没有找到相等的常量表达式，则进入表示默认情况的 default 分支开始执行，最终完成整个 switch 语句。default 关键字是可选的，如果没有 default 关键字，在程序找不到匹配的 case 分支后，则直接结束 switch 条件选择语句的执行，如图 4-2 所示。

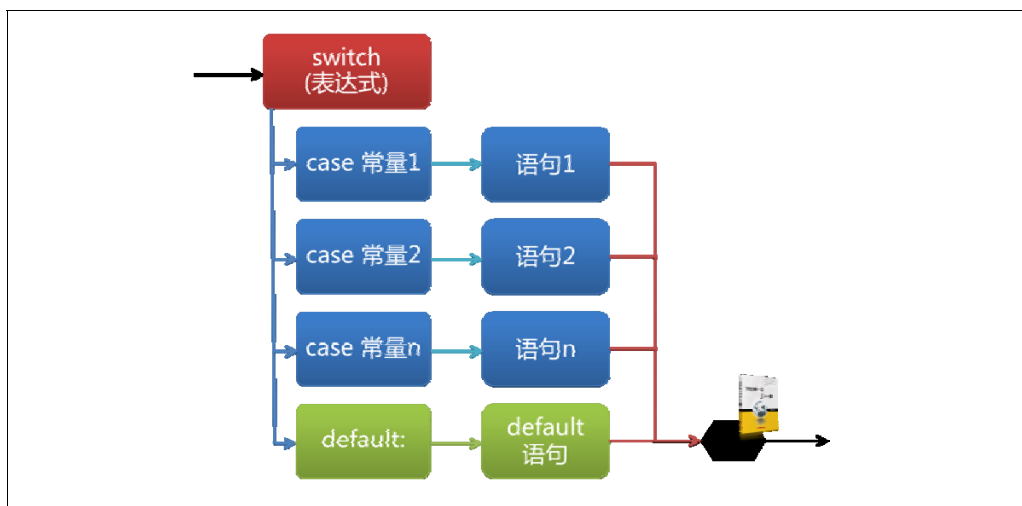


图 4-2 switch 语句的执行流程

4.3.1 while 循环

在自然语言中，有这样一种表达循环反复的句式，如下：

只要……，就一直……

在现实生活中，有很多这样的例子：只要给我加工资，我就一直好好干；只要天还没有黑，你就得一直给我干活。这些循环反复的情形都可以用 while 循环来表达。在 C++ 中，while 循环结构的语法格式如下：

```
while(条件表达式 )  
{  
    循环体语句;  
}
```

其中，条件表达式就是这个循环是否继续进行的条件，而循环体语句就是这个循环所做的事情。这样，while 循环控制语句就跟自然语言的句式很好地对应起来了。while 语句首先会判断条件表达式的值，如果表达式的值为 true，则执行循环体语句；如果为 false，则结束 while 语句。当循环语句执行完一次时，会再次判断表达式的值，根据表达式的值决定是否要进行下一次循环，如此不断循环，直到表达式的值为 false，循环结束。我们可以把 while 理解成自然语言的“只要”的意思，只要条件成立，循环就不断执行循环体语句。当条件不再满足时，就结束 while 循环控制语句。while 循环控制语句的执行流程可以用图 4-3 表述。

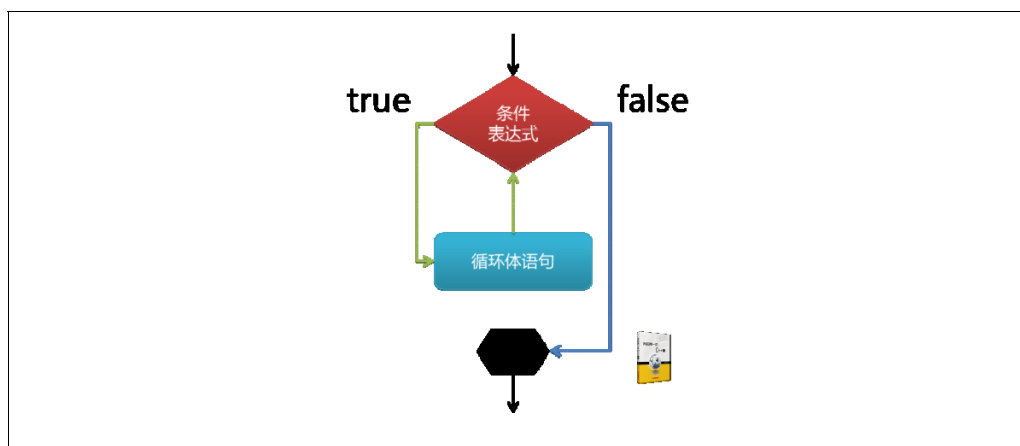


图 4-3 while 循环的执行流程

4.3.2 do...while 循环

在以上 while 循环的例子中，我们注意到，nInput 需要给定初始值才可以完成整个循环。很多情况下，while 循环的条件没有合适的初始值，那么有没有办法可以改进上面的设计呢？有，C++提供了 while 循环的孪生兄弟—— do...while 循环来解决这个问题。在 C++中，do...while 循环控制语句的语法格式如下：

```
do
{
    循环体语句；
}
while ( 条件表达式 );
```

虽然是孪生兄弟，但是 do...while 循环语句跟 while 循环语句不仅在形式上有差别，一个条件表达式在前，一个条件表达式在后，而且在执行顺序上两者也有差异。do...while 循环语句首先会执行一次循环体语句，然后再判断条件表达式的值。如果条件表达式的值为 true，则继续执行循环体语句；如果条件表达式的值为 false，则结束整个循环。do...while 循环语句的执行流程如图 4-4 所示。

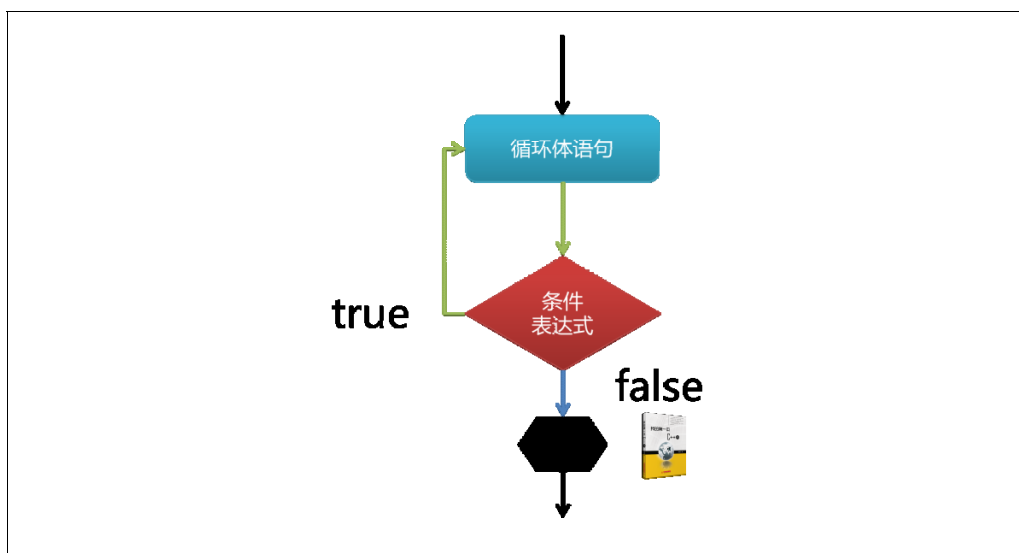


图 4-4 do...while 循环结构的执行流程

3. 更改语句

在各种循环结构中，总是有一个循环控制变量用来构成循环是否继续执行的条件。例如前面例子中的 `nInout` 就是一个循环控制变量，可以用它的值来判断是否需要进行一次循环。既然是表示循环的条件，就需要在循环中对这个变量进行修改，以反映循环的执行情况，根据执行情况决定循环是否继续进行。例如将用户输入的值赋值给 `nInput`，就是对循环控制变量的修改。`for` 循环是将循环控制变量的修改独立出来放到了更改语句中来进行。

在理解了 `for` 循环的三个要素之后，再来理解 `for` 循环的执行流程就比较清楚了。程序进入 `for` 循环语句之后，首先会执行初始化语句，完成必要的初始化工作。然后再计算条件表达式的值，如果条件表达式的值为 `true`，则执行循环体语句，再执行更改语句，修改循环控制变量。接着又开始计算条件表达式的值，根据其值决定是否需要继续下一次循环：如果条件表达式的值为 `true`，则继续下一次循环；反之，则结束整个循环的执行。`for` 循环控制结构的执行流程如图 4-5 所示。

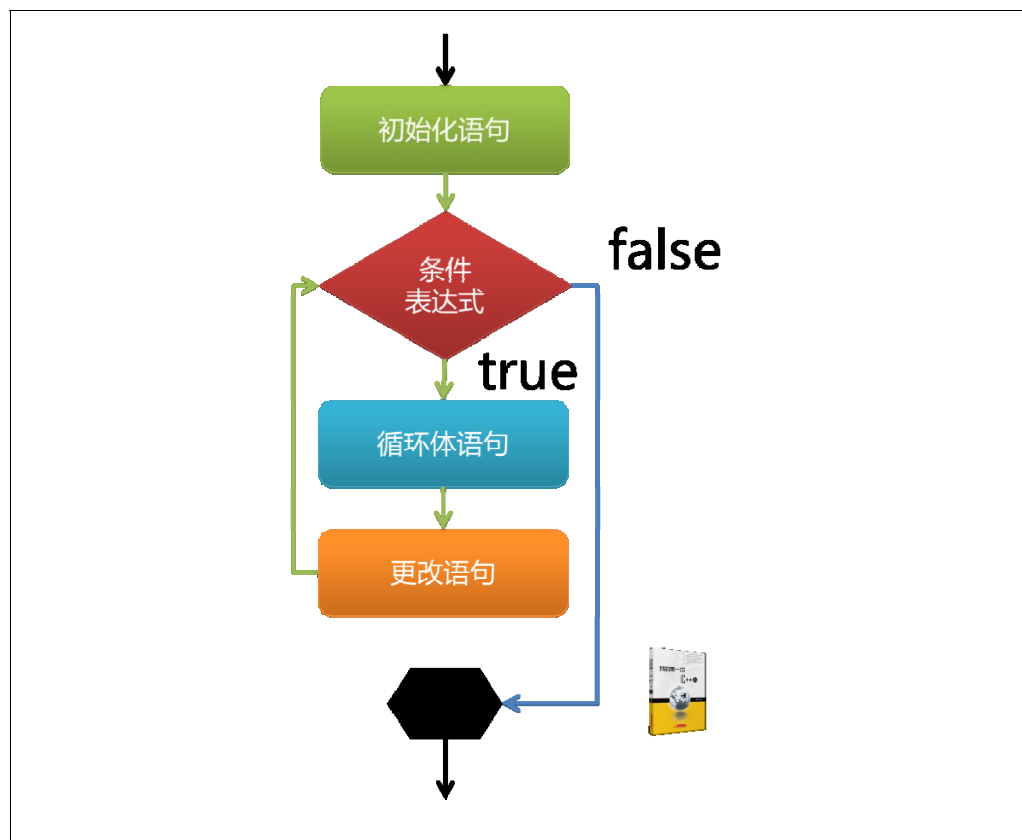


图 4-5 `for` 循环的执行流程

4.3.4 对循环进行控制：break 与 continue

```
// 大款的收支统计程序

int nTotal = 0;
int nInput = 0;
do
{
    cout<< "请输入你的收入或支出: ";
    cin>>nInput;
    if( 1000< nInput ) // 毛毛雨啊, 就不用统计了
        continue;
    nTotal += nInput;
}while( 0 != nInput );
```

在这个大款的收支统计程序中，nInput 接收用户输入后判断其值是否小于 1 000，如果小于 1 000，则执行 continue 关键字，跳过后面的加和语句“nTotal += nInput;”，而直接跳转到对条件表达式“0 != nInput”的计算，判断是否可以开始下一次循环。值得注意的是，在 for 循环中，执行 continue 后，控制条件变化的更改语句并没有被跳过，仍然将被执行，然后再计算条件表达式，尝试下一次循环。

虽然 break 和 continue 都是在某种条件下跳出循环，但是两者有本质的差别：break 是跳出整个循环，立刻结束循环语句的执行；而 continue 只跳出本次循环，继续执行下一次循环。图 4-6 展示了 break 和 continue 之间的区别。

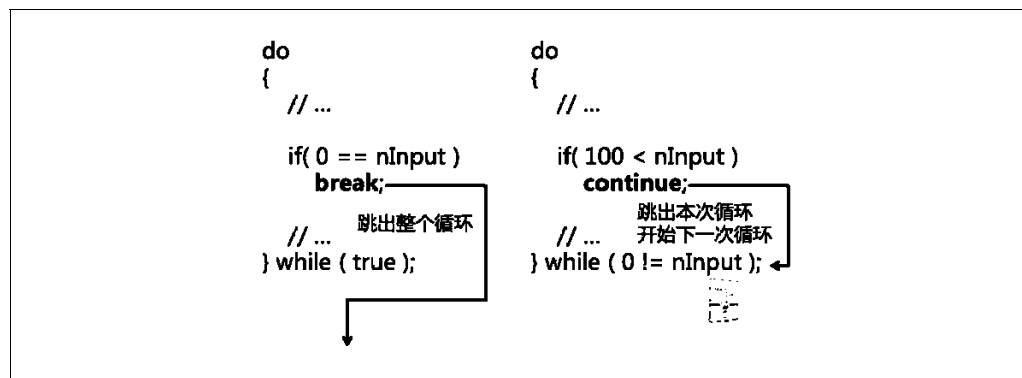


图 4-6 break 和 continue 之间的区别

· 把程序装进箱子：用函数封装程序功能

在完成豪华的工资统计程序之后，我们信心倍增，开始向 C++ 世界的更深远处探索。

现在，可以使用各种数据类型和程序流程控制结构来编写完整的程序了。但是，随着要处理的问题越来越复杂，程序的代码也越来越复杂，主函数也越来越长了。这就像我们将所有东西都堆放到一个仓库中，随着东西越来越多，仓库慢慢就被各种东西堆满了，显得杂乱无章，管理起来非常困难。面对一个杂乱无章的仓库，聪明的仓库管理员提供了一个很好的管理办法：将东西分门别类地装进箱子，然后有序地堆放各个箱子。

这个好方法也可以用到程序设计中，把程序装进箱子，让整个程序结构清晰。



5.1 函数就是一个大箱子

当要处理的问题越来越复杂，程序越来越庞大的时候，如果把这些程序代码都放到主函数中，将使得整个主函数异常臃肿，这样会给程序的维护带来麻烦。同时，要让一个主函数来完成所有的事情，几乎是一个不可能完成的任务。在这种情况下，可以根据“分而治之”的原则，按照功能的不同将大的程序进行模块划分，具有相同功能的划分到同一个模块中，然后分别处理各个模块。函数，则成为模块划分的基本单位，是对一个小型问题处理过程的一种抽象。这就像管理一个仓库，总是将同类的东西放到同一个箱子中，然后通过管理这些箱子来管理整个仓库。在具体的开发实践中，我们先将相对独立的、经常使用的功能抽象为函数，然后通过这些函数的组合来完成一个比较大的功能。举一个简单的例子：看书看得肚子饿了，我们要泡方便面吃。这其实是一个很复杂的过程，因为这一过程中我们先要洗锅，然后烧水，水烧开后再泡面，吃完面后还要洗碗。如果把整个过程描述在主函数中，那么主函数会非常复杂，结构混乱。这时就可以使用函数来封装整个过程中的一些小步骤，让整个主函数简化为对这些函数的调用，如图 5-1 所示。

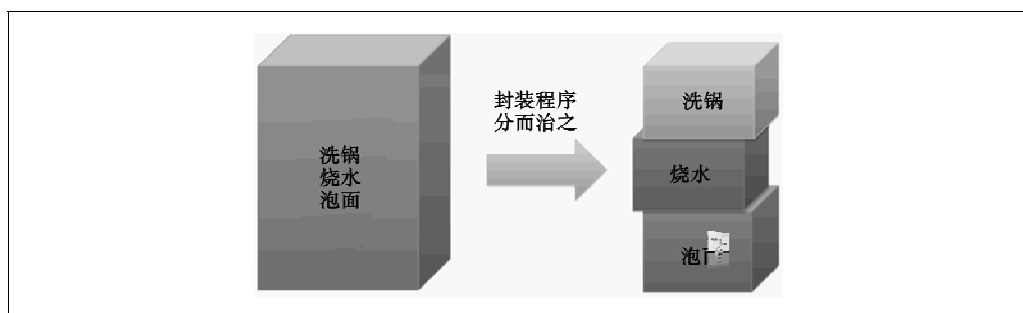


图 5-1 将程序封装到箱子，分而治之

5.1.1 将程序装到箱子中：函数的声明和定义

提问：把大象装到冰箱中需要几步？

回答：需要三步。第一，打开冰箱；第二，把大象放进冰箱；第三，关上冰箱。

提问：那么，把一个程序放进箱子需要几步？

回答：需要两步。第一，声明一个函数；第二，定义这个函数。

没错，把一个函数放进箱子比把大象放进冰箱还要简单。当分析一段长的程序代码时，往往会发现一些代码所实现的功能相对比较独立。我们将程序中这些相对比较独立的功能代码组织到一起，用函数对其进行封装，也就是将一个较长的程序分放到各个函数箱子中。

要装东西，先得准备好箱子。为了找到具体功能实现代码的箱子，需要给箱子贴上标签，这个标签就是函数的声明，如图 5-2 所示。

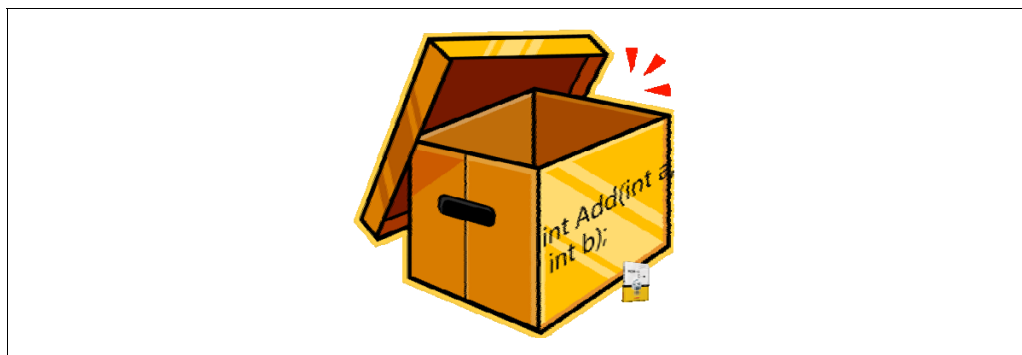


图 5-2 声明一个函数，为箱子贴上

5.1.2 函数调用机制

在学习编写函数之前，我们首先要了解函数的调用机制，学会如何调用一个已经存在的函数。世界上已经有很多函数，我们可以直接调用这些函数来完成日常任务。世界上已经有很多轮子，我们没有必要再去发明更多同样的轮子，只需要用好它们就可以了。在实际的开发中，可供调用的现有函数主要有编译器提供的库函数、Windows API 及第三方提供的函数库等。通过调用他人的函数，可以复用他人的开发成果，在其开发成果的基础上，实现快速开发，如图 5-3 所示。

有了别人提供的函数，就可以调用这些函数来完成自己的功能。两个函数之间的关系是调用与被调用的关系，我们把调用其他函数的函数称为主调函数，被其他函数调用的函数称为被调函数。一个函数是主调函数还是被调函数并不是绝对的，要根据其所处的相对位置而定：如果一个函数内部有函数，则相对其内部的函数它就是主调函数；如果它的外部有函数，则相对其外部函数它就被调函数。

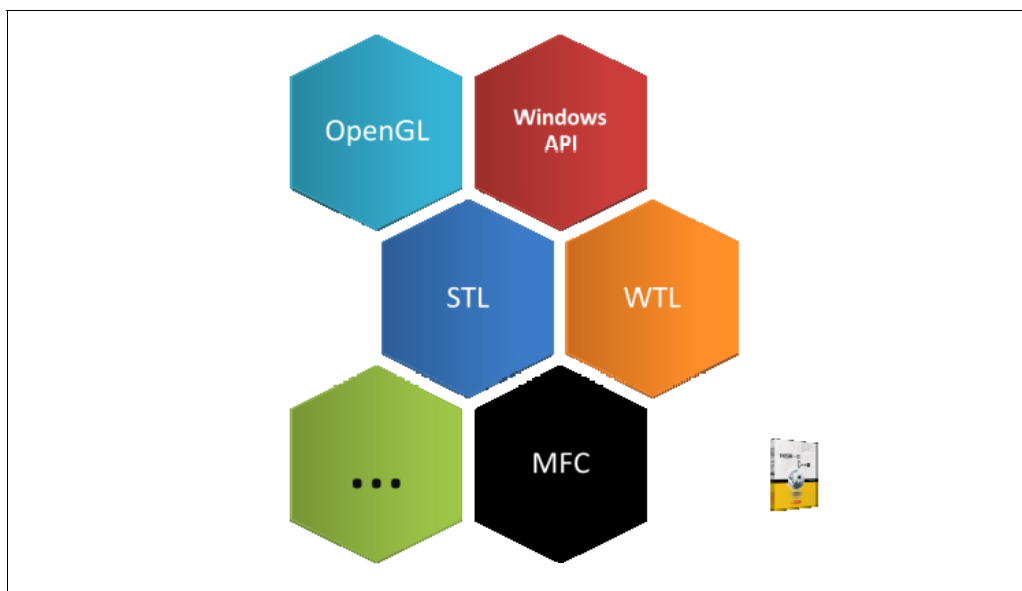


图 5-3 天上掉下个函数库

这是表示_tmain()函数对实现加法运算的 Add()函数的调用。当运行程序时，从_tmain()函数开始，首先定义 a、b 两个变量并对其赋值，然后以 a 和 b 为参数调用 Add()函数来计算这两个数的和。虽然从表面上我们并不能看到函数调用的实现细节，但是在背后函数调用却做了很多事情。首先用 a 和 b 对 Add()函数的两个形式参数赋值，将控制权交给 Add()函数，开始执行 Add()函数。在 Add()函数中，两个形式参数 a 和 b 经过赋值后，其值分别为 1 和 2，这就完成了从主调函数传递数据给被调函数的过程。然后 Add()函数开始执行具体的运算过程，也就是计算两个形式参数的和。最后还需要用 return 关键字将计算结果返回，作为整个函数调用表达式的结果，主调函数可以利用这个结果进行进一步的赋值或者运算。被调函数返回后，控制权重新交还给主函数。主函数继续执行，将函数的返回值赋值给 nResult 变量，并将计算结果输出。整个调用过程如图 5-4 所示。

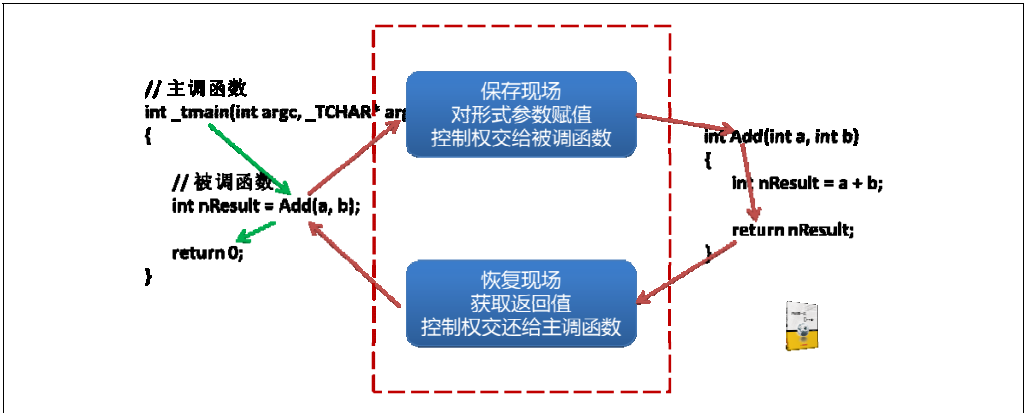


图 5-4 函数调用的执行流程

在图 5-4 中，箭头的方向代表整个 Add()函数被调用的流程，虚线框包围的部分是系统为了实现函数调用而额外做的幕后工作。

为了解函数的嵌套调用，下面来看一个计算平方和的程序。

```
// 计算平方函数
int Power( int n )
{
    return n*n;
}
// 计算平方和函数
int PowerSum( int a, int b )
{
    return Power(a) + Power(b);
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 调用求平方和函数
    int nResult = PowerSum(2,3);

    return 0;
}
```

我们知道，求平方和的方法是先求两个数的平方，然后再进行加和运算以求得两个数的平方和。按照这样的思路，我们首先将求平方和的任务分解为求平方及求和两个子任务，也就是 `Power()` 和 `PowerSum()` 两个子函数。在程序中执行具体的计算过程的时候，在主函数中调用 `PowerSum()` 函数，而 `PowerSum()` 函数又套嵌调用 `Power()` 函数，这样就将一个比较复杂的问题通过不断细化和分解，最后转化为比较简单的问题。这正好也反映了程序设计中的“自顶向下，逐步求精”的设计思想。

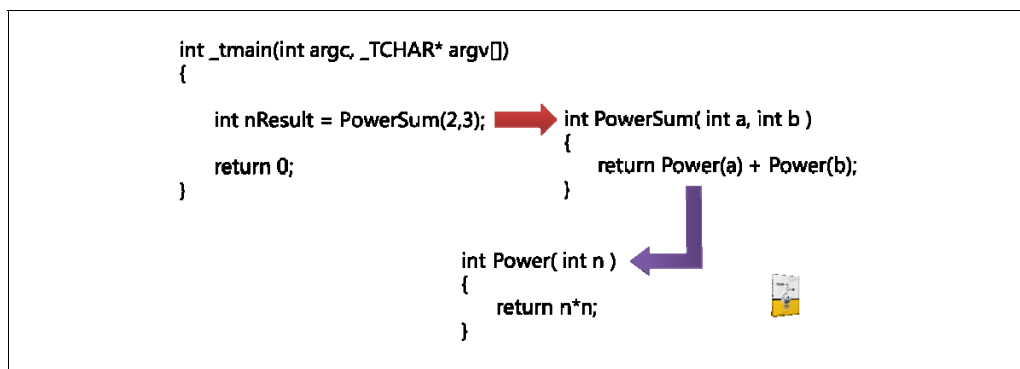


图 5-5 平方和程序的函数套嵌调用

5.1.4 把数据放到箱子中：函数参数的传递

我们可以把东西放进箱子中，同样，通过函数参数的传递，也可以将数据放到函数箱子中。我们知道，函数是用来完成某个相对独立功能的模块。函数在完成这个功能的时候，往往需要外部数据的支持，这时就需要在调用这个函数时向它传递所需要的数据。例如，当调用一个加法函数时，需要向它传递两个数作为加数和被加数。在定义一个函数的时候，如果这个函数需要跟外部进行数据交换，就会在函数定义中加入形式参数表，通过形式参数表可以将数据从函数外部传入函数内部。例如，可以这样定义一个加法函数：

```
// 声明并定义 Add 函数
// 用形式参数表来表示这个函数需要跟外界进行数据传递
int Add( int a, int b )
{
    return a + b;
}
```

从 Add()函数的声明中可以知道这个函数需要两个整型数作为参数，所以可以在调用的时候给它传递两个整型数作为参数：

```
// 以 1 和 2 作为参数调用 Add() 函数
// 也就是向这个函数传递 1 和 2 这两个数据
int nResult = Add(1, 2);
```

我们将在函数调用时给出的参数称为实际参数，例如 1 和 2 就是实际参数。在进行函数调用的时候，系统会使用调用函数时给出的实际参数分别对函数声明中的各个形式参数进行赋值。例如，这里 1 和 2 两个参数分别会赋值给 Add()函数的两个形式参数 a 和 b，也就是说，在 Add()函数内部，a 和 b 两个变量的值这时就是 1 和 2，这样就通过实际参数和形式参数实现了从函数外部向函数内部传递数据，如图 5-6 所示。

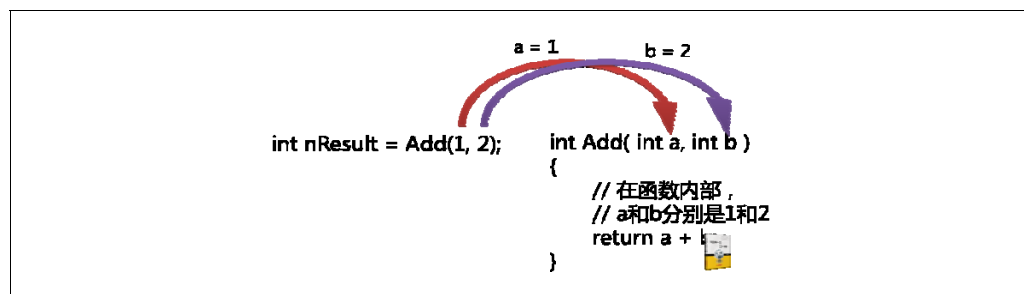


图 5-6 函数调用过程中的参数传递

```

// 输入员工的工资数据到工资数组中

int InputSalary(int* pSalary, const int MAX_NUM )
{
    int nTemp = 0; // 临时变量，暂存用户输入的数据
    int nIndex = 0; // 输入的序号
    do
    {
        cout<<"请输入员工"<<nIndex<<"的工资: "<<endl;
        cin>>nTemp;
        // 如果输入的是负数，表示输入工作结束，跳出循环
        if ( nTemp < 0 )
        {
            break;
        }

        // 将合法的数据保存到数组中，开始下一次输入
        // 通过修改传入的数组，实现数据的传出
        pSalary[nIndex] = nTemp;
        ++nIndex;
    } while ( nIndex< MAX_NUM );

    // 返回输入的数据总数
    return nIndex;
}

```

InputSalary()函数用来输入员工的工资数据。我们利用指针向这个函数传递一个数组，并在函数内部将合法的输入保存到数组中，修改数组的数据。这样函数外部的数组就保存了函数内部的数据，间接实现了函数内部数据的传出。另外在这个函数中，还利用函数返回值从函数中得到了所有输入的数据总数。也就是说，利用函数返回值和函数参数从函数中传出数据这两种方式是可以同时使用的。当然，也可以使用 void 作为函数的返回值类型，从而使用函数的指针参数来完成数据的传入、传出，如图 5-7 所示。

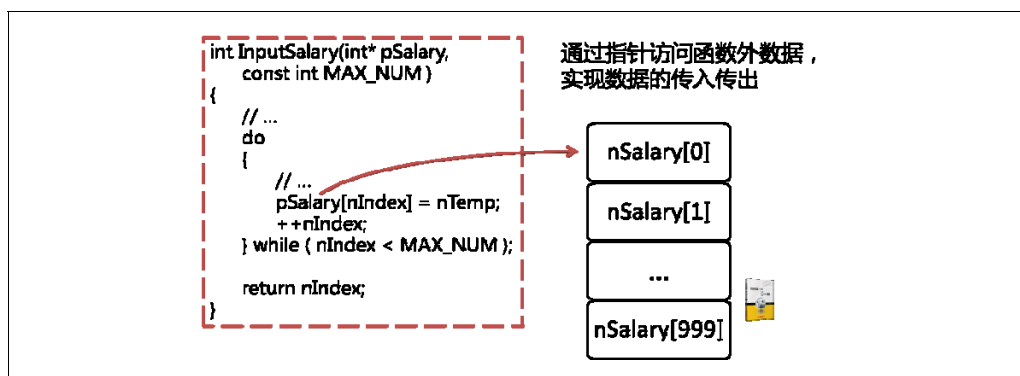


图 5-7 通过指针访问函数外部数据

下面来看看如何使用函数重载来解决上文中 Add()函数不支持浮点型数的问题。

```
// 计算整型数和的 Add()函数
int Add( int a, int b )
{
    cout<<"int Add( int a, int b )被调用! "<<endl;
    return a + b;
}
// 计算浮点型数和的 Add()函数
float Add( float a, float b )
{
    cout<<" float Add( float a, float b )被调用! "<<endl;
    return a + b;
}

int _tmain(int argc, _TCHAR* argv[])
{
    // 因为参数是整型数
    // 调用 int Add( int a, int b )
    int nSum = Add(2,3);
    cout<<" 2 + 3 = "<<nSum<<endl;

    // 因为参数是浮点数
    // 调用 float Add( float a, float b )
    float fSum = Add(2.5f,3.7f);
    cout<<" 2.5 + 3.7 = "<<fSum<<endl;

    return 0;
}
```

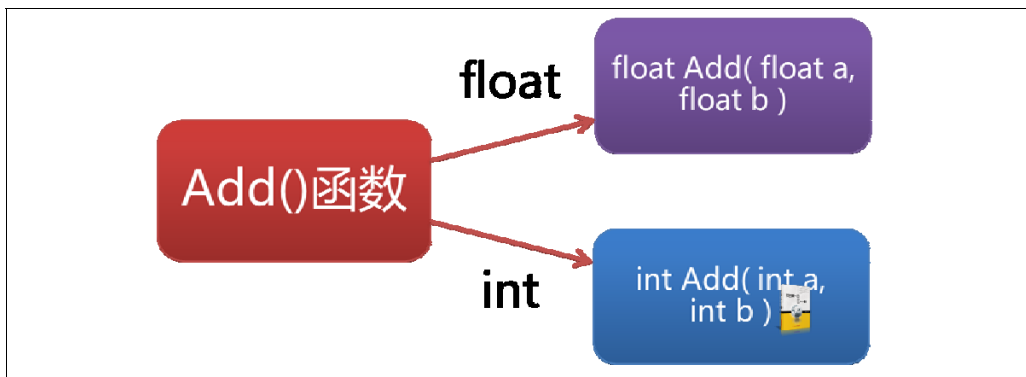


图 5-8 同样的函数名，不同的函数实现

5.4.1 函数声明的设计规则

函数的声明，也称为函数的接口，它是函数跟外界打交道的通道。它就像函数箱子上的标签一样，可通过该标签了解箱子中封装的是什么功能，需要什么样的输入数据，以及能够返回什么样的结果。大量实践表明，一个函数是否好用，往往由其接口设计的好坏决定。在设计实现函数的时候，不仅要让函数的功能正确，还要让函数的接口清晰明了，有较高的可读性。只有这样，在使用这个函数的时候，才会清楚函数的功能及函数的输入/输出参数等，从而正确使用这个函数。如果函数的接口不清楚，则很容易造成函数的误用。

在函数接口的设计上，通常有如下几种规则。



图 5-9 优秀函数的五项修炼

· 当 C++ 爱上面向对象

很多第一次进入 C++ 世界的人都会问：C++ 中的两个加号到底是什么意思啊？

我们知道，C++ 语言是从 C 语言发展起来的，C++ 比 C 多出的两个加号，实际上是 C++ 的自增操作符，表示 C++ 语言是在 C 语言的基础上添加了新的内容。如果说其中一个加号代表 C++ 在 C 的基础上增加了模板、异常处理等现代程序设计语言的新特性，那么另外一个加号则代表 C++ 支持面向对象程序设计思想。正是这两个加号，让 C++ 语言与 C 语言有了最本质的区别，尤其是其中的面向对象程序设计思想，使得 C++ 完成了从 C 语言到 C++ 语言的进化，从而让 C++ 语言既具备了 C 语言的优秀根基，又能够体现现代程序设计语言的发展趋势，使得 C++ 在多种程序设计语言中经久不衰。

武林中流传这样一句话：“平生不识陈近南，纵称英雄也枉然！”这句话说明了陈近南的名声之响。如果说陈近南是武林中响当当的人物，那么面向对象程序设计思想则是程序设计界名副其实的老大。可以毫不夸张地说，作为一名程序员，如果不知道面向对象程序设计思想，纵称高手也枉然。

既然面向对象程序设计思想的名声如此之大，那么它到底是怎么回事？它跟 C++ 有着怎样的爱恨情仇？它为何会受到这么多人的追捧和欢迎？别着急，且听我一一道来。



6.1.1 自顶向下，逐步求精：结构化程序设计

在学习新的面向对象程序设计思想之前，先来“忆苦思甜”一下，看看在面向对象程序设计思想出现之前的软件是如何设计和开发的。回顾前面章节中曾经学习过的例子：总是先提出问题；然后分析问题的处理流程；接着根据需要把一个大的问题划分为几个小的问题，分成各个子模块；其次解决每个小问题，实现每个模块；最后通过主函数按照某种次序调用这些模块，组织业务逻辑流程，最终解决问题。像这样从问题出发，自顶向下，逐步求精的开发方法，称为“结构化程序设计思想”，如图 6-1 所示。

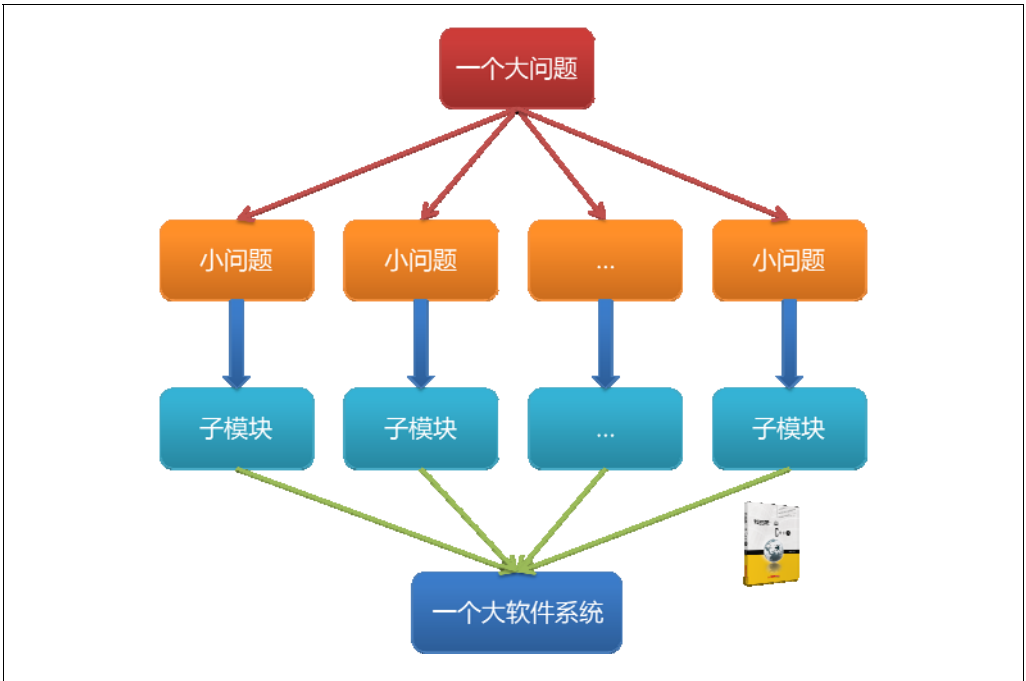


图 6-1 结构化程序设计的流程

6.1.2 万物皆对象：面向对象程序设计

面向对象程序设计（object oriented programming, OOP）是对结构化程序设计的继承和发展，它不但汲取了结构化程序设计的精华，而且以一种更接近人类思维方式的形式来分析和解决问题：程序是对现实世界的描述，现实世界的基本单元是物体，程序中的基本单元就是对象。

面向对象程序设计思想认为：现实世界是由很多彼此相关并互通信息的实体——对象（object）组成的。大到一个星球、一个国家，小到一个人、一个分子，无论是有生命的，还是没有生命的，都可以看成一个对象。通过分析这些对象，发现每个对象都由两部分组成：描述对象状态或属性的数据，以及描述对象行为或功能的函数（方法）。结构化程序设计思想将数据和函数分开，而面向对象程序设计却将数据和函数紧密结合，共同构成对象来更加准确真实地描述现实世界。这可以说是两者最本质的区别。

跟现实世界相对应的，在面向对象程序设计思想中，我们用对象来代表现实世界中的实体，每个对象都有自己的属性和行为，而整个程序由一系列相互作用的对象构成，对象之间通过互相发送消息来完成互通信息，以此完成复杂的业务逻辑。比如在一个学校中，可以利用面向对象思想将老师和学生这两种实体抽象成对象，他们的属性有部分相同的，比如姓名、年龄等，而有部分属性是不同的，比如老师有职务这个属性，而学生则没有。另外，老师和学生这两种对象还有各自不同的行为，比如老师有备课、上课、批改作业的行为，而学生则有听课、完成作业等行为。老师和学生各自负责自己的行为 and 职责，同时又相互发生联系，老师上课，学生听课。通过对象之间的相互作用，学校才能正常运作。整个流程如图 6-2 所示。

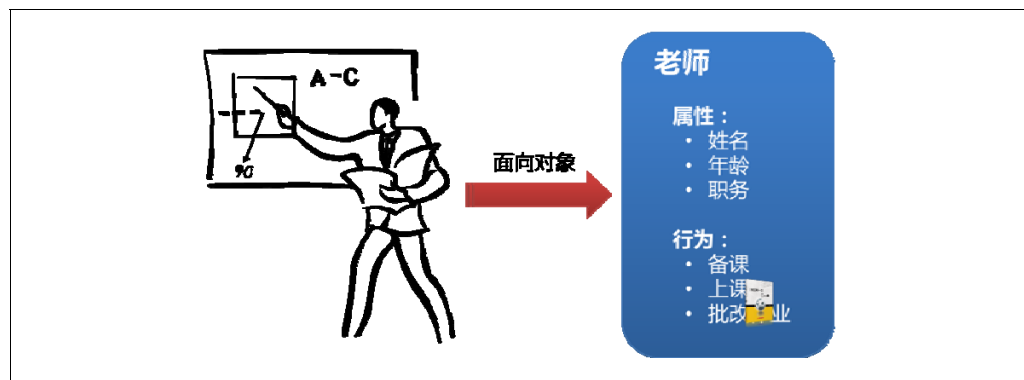


图 6-2 用面向对象思想将老师抽象成对象

6.1.3 面向对象的三座基石：封装、继承与多态

我们知道，面向对象程序设计思想是在传统的结构化程序设计思想基础上发展起来的。那么，是哪些特点让面向对象程序设计思想从根本上区别于结构化程序设计思想，使其成为一种全新的程序设计思想？通常认为，封装、继承和多态是面向对象思想的三座基石，它们共同构成了面向对象思想的核心特征，如图 6-3 所示。

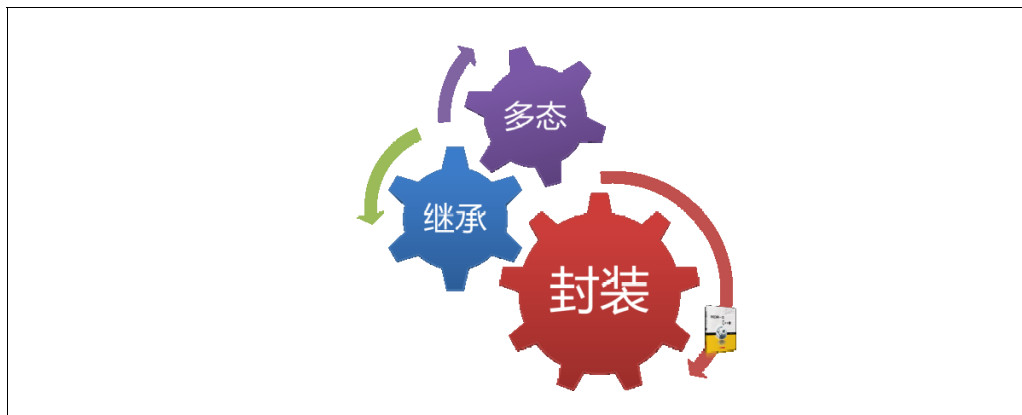


图 6-3 面向对象思想的三座基石

1. 封装

我们知道，程序是用来描述现实世界的，那么在现实世界中，又是如何描述周围的物体的呢？我们总说某个物体是什么，能做什么。这就是我们描述物体所必需的数据和算法。

在传统的结构化程序设计思想中，程序中的数据和算法是相互分离的。也就是说，在描述一个物体的时候，物体是什么（数据）和物体能做什么（算法）是相互分开的。但是在面向对象思想中，它是通过封装机制将数据和相应的算法捆绑到一起的，形成一个完整的、同时具有属性（数据）和行为（算法）的对象，避免了外界的干扰和不确定性。简单来说，对象就是封装数据和操作这些数据的算法的逻辑实体，也是现实世界中物体在程序中的反映，如图 6-4 所示。

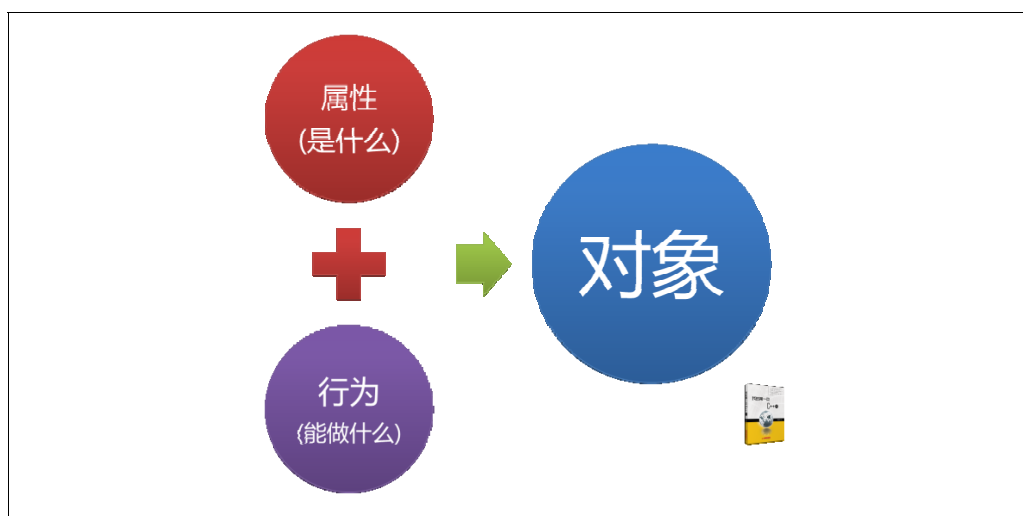


图 6-4 将数据和算法封装封装成对象

2. 继承

继承是可以让某个类型的对象获得另一个类型的对象的属性的方法。就如同现实世界中的等级进化一样，语文老师属于老师这个类别，具有了老师这个类别的共有属性；而老师又属于人类这个类别，具有了人类的共有属性。这种继承的原则，可以让每个子类都轻松具有父类的公共特性。

正是这种从父类继承属性的特点，可以很好地支持代码的重用。比如，想给一个已有的类别增加新的属性，而又不想改变这个类别。利用继承，就可以通过从这个已存在的类别派生一个新的类别来实现。这个新的类别将具有原来类别的特性和新添加的特性。而继承机制的强大魅力就在于它允许我们充分利用已经存在的类别（接近需要，而不是完全符合需要的类），同时又可以以某种方式修改这个类别，添加新的特性，而不会影响其他的東西，如图 6-5 所示。

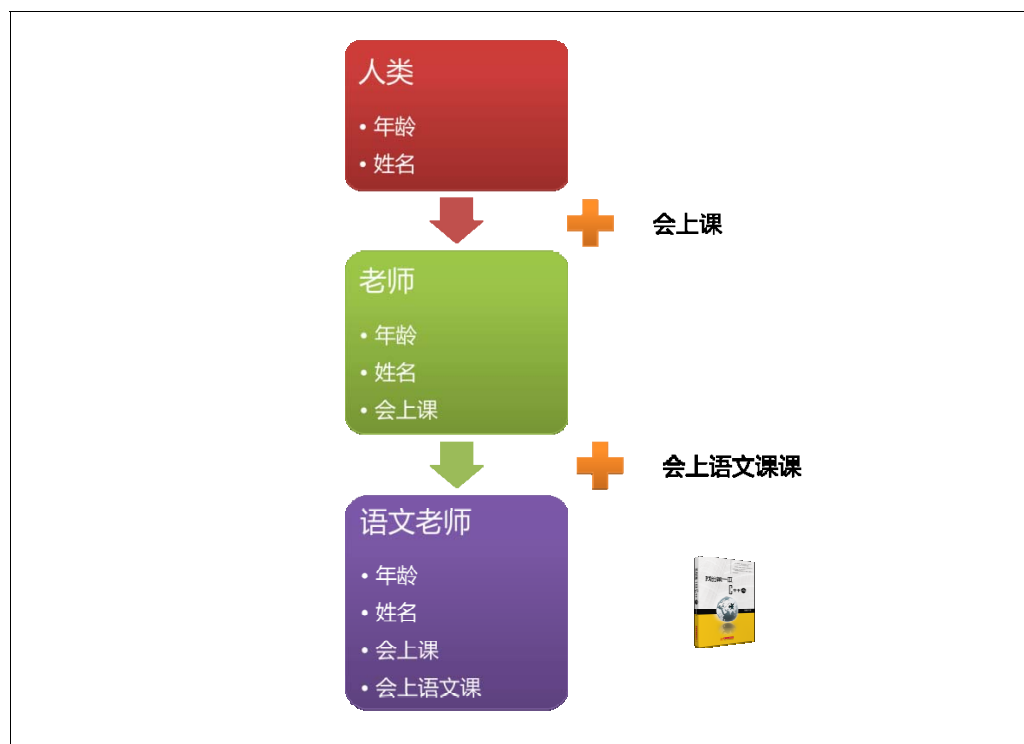


图 6-5 继承

3. 多态

见人说人话，见鬼说鬼话，这是说一个人两面三刀，不是什么好人。可在 C++ 中，这种在不同情况下做不同事情的现象，却冠以一个冠冕堂皇的名字——多态，成为面向对象思想的一个重要特征。

多态，就是指对象在不同情况下具有不同形式的能力。对于不同的对象，某个操作可能会有不同的行为，这依赖于这个动作所要操作数据的类型。比如说加和操作，如果操作的数据是数，它就是对两个数求和；如果操作的数据是两个字符串，则它将连接两个字符串。这就是 C++ 世界中的“见人说人话，见鬼说鬼话”。

多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以相同的方式予以调用。简单来说，多态机制允许通过相同的接口引发一组相关但不相同的动作，通过这种方式，可以减少代码的复杂度。在某个特定的情况下应该作出怎样的动作由编译器决定，而不需要程序员手工进行干预，为程序员省了很多事，如图 6-6 所示。

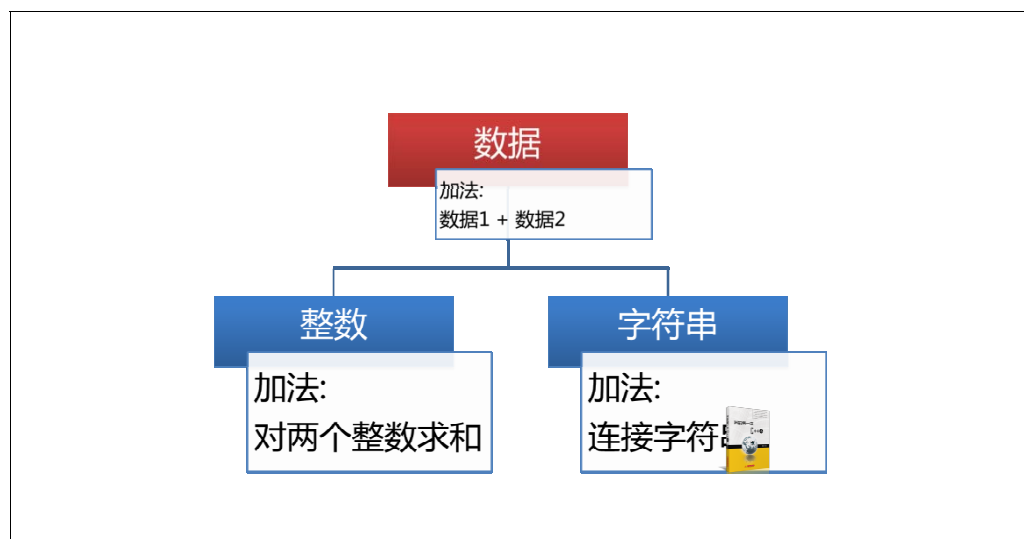


图 6-6 多态

6.2.1 类的声明和定义

其中，`class` 是 C++ 中声明类的关键字，其后跟类的名字，通常用一个名词来描述这个类所代表的一类对象，再后面是类的继承方式和基类的名字，如果没有基类，则这一部分可以省略。

完成类的名字及继承关系的定义后，可以开始在类的主体中描述这个类的属性和行为。我们知道，类实际上是对多个同类型对象的抽象。通过面向对象的封装机制，可将现实世界中的物体封装成对象，这个对象就包含了这个物体的属性和行为。在 C++ 中，我们总是用一些变量来代表某些数据，将变量引入类的声明之中成为类的成员变量，那么这些变量就成为对对象属性的描述。比如，一个老师的年龄，可以在类中添加一个整型变量来描述；他的姓名，可以在类中使用一个字符串型变量来描述。通过这些成员变量，可以描述一个对象的所有属性。

除了对象的属性之外，对象的另外一个重要组成部分就是它的行为。在 C++ 中，我们用函数来描述一个行为动作。同样，我们也将函数引入类之中成为它的成员函数，用来描述类对象的行为。比如，一个老师有备课的行为动作，我们就可以为老师这个类添加一个 `PrepareLesson()` 函数，在这个函数中可以对老师备课动作进行具体的定义。类的构成如图 6-7 所示。

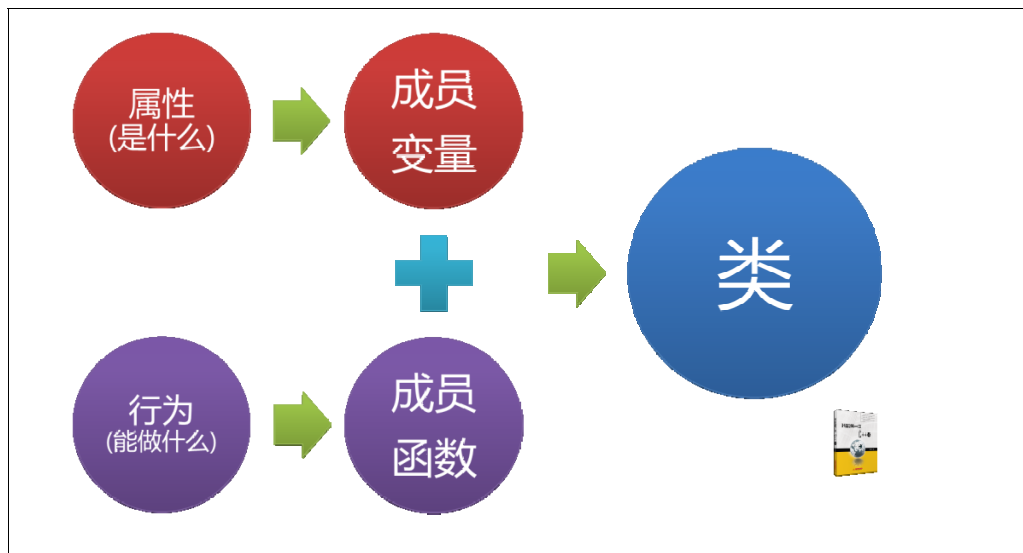


图 6-7 类的构成

6.2.2 使用类创建对象

也许大家会问：类和对象的关系是怎么样的？为什么要定义一个类，然后用这个类来定义对象？在 C++ 中，我们是用类的概念来反映面向对象思想的。类是一种抽象机制，它描述的是同一类对象所共有的属性和行为。比如，学校里有多位老师，老师这个类描述的是这多位老师，也就是多位老师这类对象所共有的属性和行为。反过来，类的对象就是该类的一个特定实体，比如学校某个老师就是 Teacher 这个类的一个特定实体，也就是这个类的对象。简单来讲，多个对象的抽象是类，类的具体化是对象。多位老师可以抽象成 Teacher 类，而陈老师就是 Teacher 类的一个具体对象了，如图 6-8 所示。

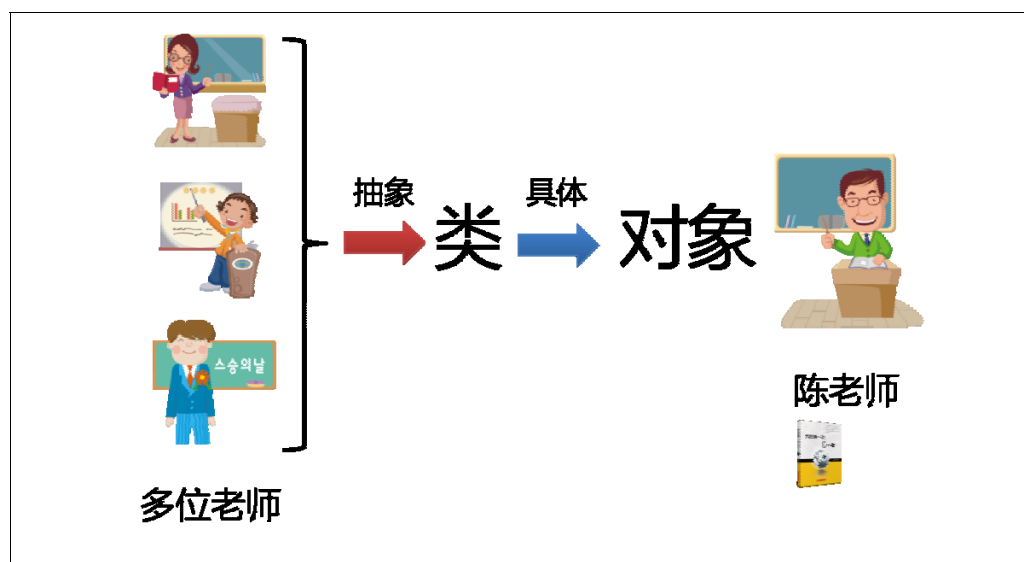


图 6-8 类与对象的关系

6.2.6 不准动我的奶酪：类成员的访问控制

类成员包括类的成员变量和成员函数，分别用来描述类的属性和行为。而类成员的访问控制决定哪些成员是公开的，可以被外界访问；哪些成员是私有的，只能在类的内部访问，外界无法访问。就像一个人盘子中的奶酪，只能自己动，别人是不能动的。

大家可能会问，为什么类这么自私自利，还要对类成员的访问加以控制，大公无私不是挺好的吗？我们可以设想这样的情形：名字是一个人的属性，当然也会是你的属性。一般而言，只有你老爸和你自己有权利修改你的名字。如果其他人随便修改你的名字，恐怕你不会乐意。所以要对对象的属性和行为的访问加以控制，以此来保护某些属性和行为不被非法访问。

在 C++ 中，对类成员的访问控制是通过设置成员的访问级别来实现的。访问级别按照访问的范围不同分成公有类型（public）、保护类型（protected）和私有类型（private）三种，如图 6-9 所示。



图 6-9 访问级别

6.3.1 从具体到一般：用类实现封装

封装好的类通过提供特定的外部接口，可让其他对象调用自己的行为完成任务，但是对外隐藏了对象行为的实现细节。另外，在大多数情况下，外界是无法直接访问对象的属性的，这样就很好地实现了对数据的隐藏，如图 6-10 所示。

抽象与封装，用来将现实世界的事物转变成 C++ 中的各个类，也就是用程序语言来描述现实世界。结构化程序设计中也有抽象这个过程，只是它的抽象仅针对现实世界中的数据，而面向对象的抽象不仅包括事物的数据，还包括事物的行为，更进一步地，面向对象利用封装将数据和行为有机地结合在一起，从而更加真实地反映现实世界。抽象与封装，完成了从现实世界中的具体事物到 C++ 中一般类的过程，是将现实世界程序化的有力武器。

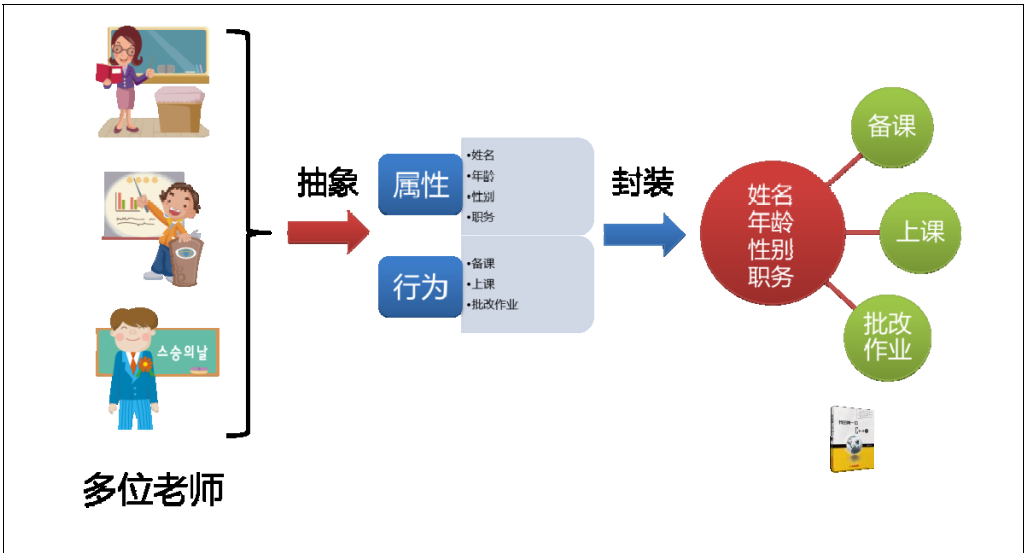


图 6-10 抽象与封装

6.3.2 子承父业：用基类和派生类实现继承

所谓继承，就是从父辈处得到父辈的属性和行为。同时，继承又不仅仅是完全照搬父辈的属性和行为，而是通过继承对父辈的属性和行为做进一步的细化或者扩充，形成新的类。这样，当复用旧有的类形成新类时，只需要从旧有的类继承，然后修改或者扩充需要的属性和行为即可。新类复用旧有对象的属性和行为，体现了面向对象的共享机制。在 C++ 中，我们把旧有的类称为基类或者父类，而把从基类继承产生的新类则称为派生类，有时也可以形象地称为子类。下面来看一个实际的例子，如图 6-11 所示。

从这棵继承树中可以看到，老师和学生都继承自人类，这样，老师和学生就具有了人类的属性和行为，而小学生、中学生、大学生继承自学生这个类，他们不但具有人的属性和行为，同时还具有学生的属性和行为。通过继承，派生类不用再去重复设计和实现基类已有的属性和行为，可以直接通过继承拥有基类的属性和行为，从而实现设计和代码最大程度上的复用。

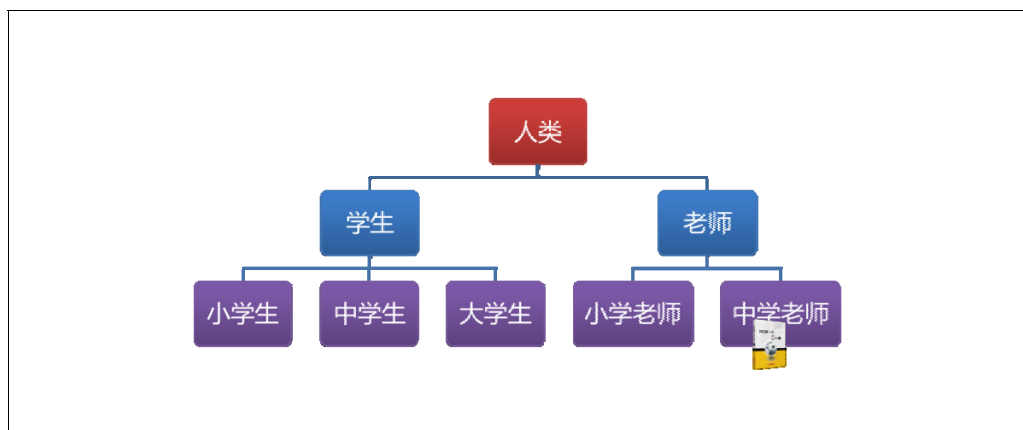


图 6-11 现实世界的继承关系

6.3.3 龙生九子，各有不同：利用虚函数实现多态

在理解了面向对象的继承机制之后，我们知道了在大多数情况下派生类是基类的“一种”，就像学生是人中的一种一样。换句话说，学生是人的一种，那么在使用“人”的时候，这个“人”可以是“学生”，而“学生”也可以应用在“人”的场合。比如可以问“教室里有多少人”，实际上问的是教室里有多少学生。这种用基类指代派生类的关系反映到 C++ 中，就是基类指针可以指代派生类的对象，而派生类的对象也可以当做基类对象使用。这样的解释对大家来说是不是很抽象呢？没关系，可以回想生活中我们是否经常会遇到这样的场景：“上车的人请买票。”在这句话中，涉及一个类——人，以及它的一个动作——买票。上车的人可能是老师、学生，也可能是工人、农民或者某个程序员，为什么售票员不说“上车的老师请买票”或者说“上车的工人请买票”，而仅仅说“上车的人请买票”就足够了呢？这是因为“人”是我们的基类，虽然上车的人可能是老师、学生、公司职员等，但是他们都是“人”这个基类的派生类，所以这里就可以用基类“人”来指代所有派生类对象，如图 6-12 所示。

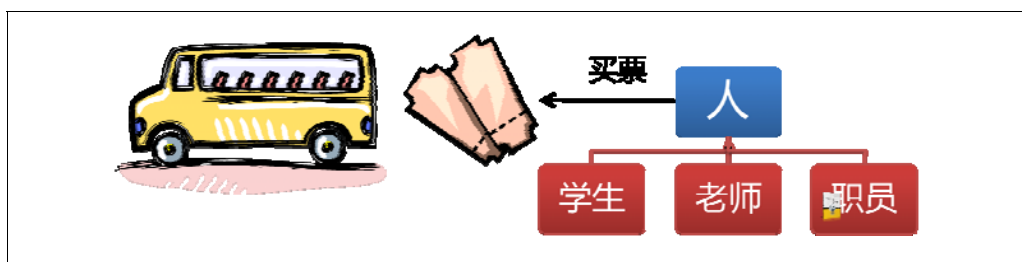


图 6-12 “上车的人请买票”

。

6.4.1 从问题描述中发现对象

问题描述中哪里有对象？我怎么没有看到？

仔细看，问题描述中的各名词实际上就是对象。

这下豁然开朗了，我们来看看问题描述中有哪些名词，也就是要寻找的对象。首先，遇到的第一个名词是工资管理系统。然后是该系统所管理的员工，因为级别的不同，员工又分为经理和普通员工，这些就构成了整个问题中的所有对象。

除了找到对象之外，还可以发现对象之间的各种关系：工资管理系统管理员工对象，它们之间是一对多的关系，同时，经理和普通员工同属于员工，它们是从员工所派生出来的。

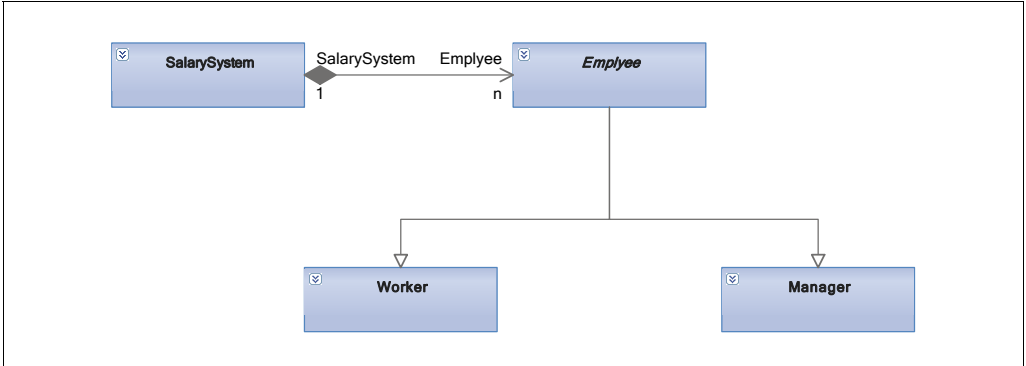


图 6-13 类以及类之间的关系

6.4.2 分析对象的属性和行为

找到对象之后，就可以分析这些对象所拥有的属性和行为，利用面向对象封装的机制将其封装成具体的类。首先，分析这个问题中最基础的员工对象。根据问题描述，我们需要输入员工的姓名和入职时间，所以这个对象必需的属性是姓名和入职时间。同时，工资管理系统要显示所有员工的信息，这也就是员工对象必须对外提供公有接口以便外界能够获得它的姓名和入职时间属性。另一方面，根据员工级别的不同，员工被分成经理和普通员工两类，根据面向对象中继承的思想，为了复用基类的设计和代码，我们让这两个类从基类中派生，直接拥有基类的属性和行为。但是，在问题描述中，经理和普通员工这两个对象的工资计算方式是不同的，所以这里可以利用面向对象的多态机制对基类中的行为进行重新定义。通过这样的分析，我们就清楚了对象的属性和行为，利用面向对象的多态机制将这些属性和行为封装起来，就形成了具体的类，如图 6-14 所示。

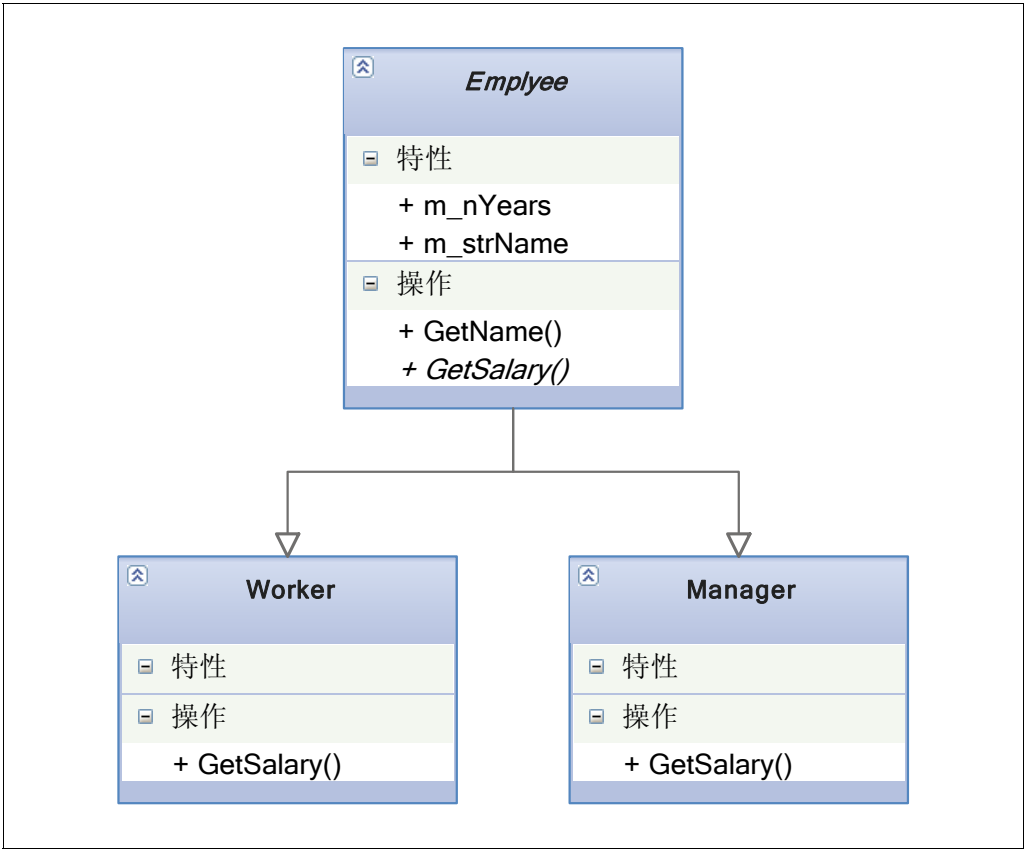


图 6-14 员工类的属性和行为

完成了员工相关类的创建后，我们来看看对员工对象进行管理的工资管理系统，分析它应该具有哪些属性和行为。根据问题描述，工资管理系统要对所有输入的员工对象进行管理，所以它必须有一个属性来保存它所管理的所有员工并且有一个属性用来记录当前的员工总数。因为要管理多个员工对象，所以可以使用数组作为其保存员工对象的属性。同时，为了处理问题描述中的事务，例如员工信息的输入和显示及计算平均工资等，它必须具备相应的行为，也就是它必须提供相应的成员函数供外界调用以完成相应的事务。经过这样的分析，工资管理系统类实现如图 6-15 所示。

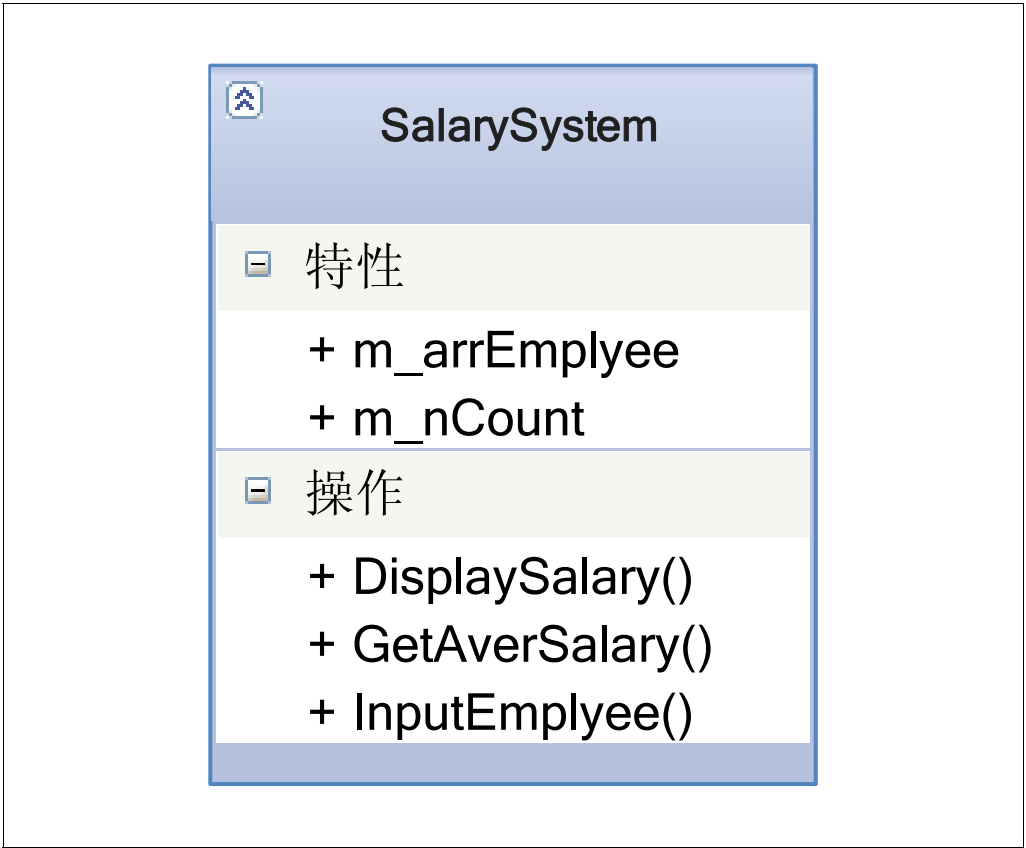


图 6-15 工资管理系统类的属性和行为

6.5.1 庖丁解牛：C++类对象的内存模型

仔细分析这份类对象的检测报告就会发现，对象的第一个成员变量的地址跟整个对象的地址相同，这表明在对象的内存模型中排在第一位的就是第一个成员变量。然后，第二个成员变量的地址就是第一个成员变量的地址加上它所占用的内存空间，换句话说，第二个成员变量紧排在第一个成员变量之后。由此可以看出，对象中的成员变量是按照类声明中的顺序依次排列的。

既然成员变量是依次排列的，那么成员函数会不会也如此呢？看看检测报告中的成员函数的地址，发现成员函数的地址是一个奇怪的内存地址。这是为什么呢？原来，跟每个对象都有一份成员变量不同，并不是每个对象都有一份成员函数。因为同一个类的所有对象的成员函数都是相同的，没有必要为每个对象都配备一份成员函数。在 C++ 类对象模型中，类的所有成员函数被放在一个特殊的位置，所有这个类的对象都共用这份成员函数。整个类对象的内存模型如图 6-16 所示。

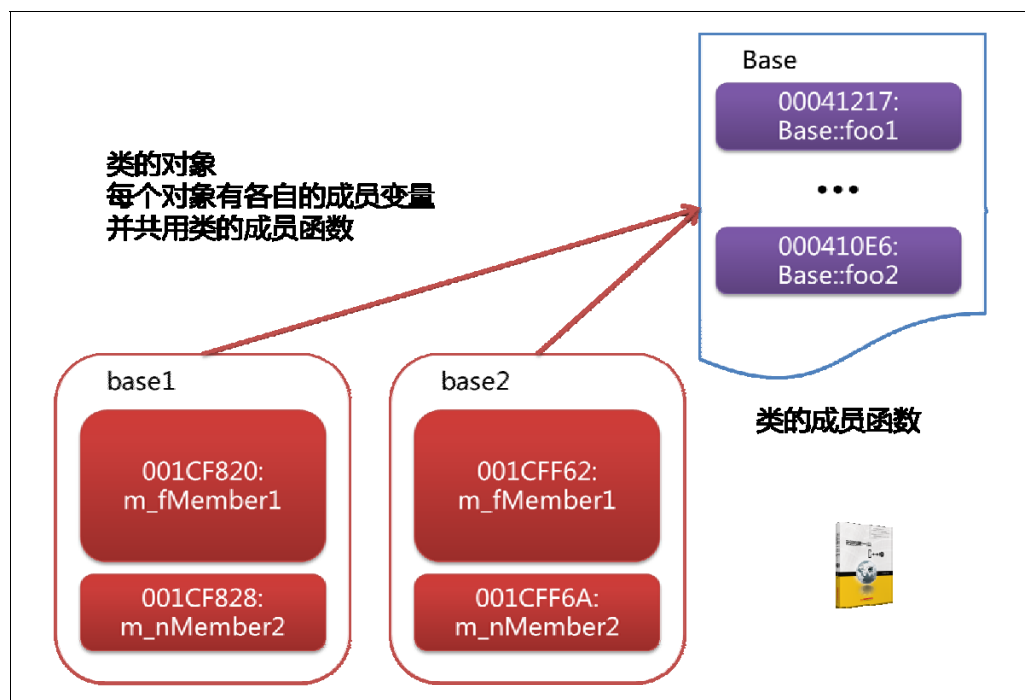


图 6-16 类对象的内存模型

6.5.2 “Who am I?": 特殊的 this 指针

从以上代码中可以清楚地看到，类成员函数中的成员变量前都有一个 `this` 指针。当通过某个对象调用它的成员函数时，系统会隐式地传递给成员函数一个指向这个对象的指针，这就是 `this` 指针。当我们在成员函数中访问类的成员变量时，这时 `this` 指针指向的是调用这个函数的对象，所以对成员变量的访问也就变成了对这个对象所属的成员变量的访问。例如，通过 `aBase` 对象调用 `SetValue()` 成员函数，那么在 `SetValue()` 成员函数中，隐藏的 `this` 指针就指向 `aBase` 这个对象。显然，成员函数中的“`this->m_nVal = nVal`”语句也就是对 `aBase` 的 `m_nVal` 成员变量赋值，如图 6-17 所示。

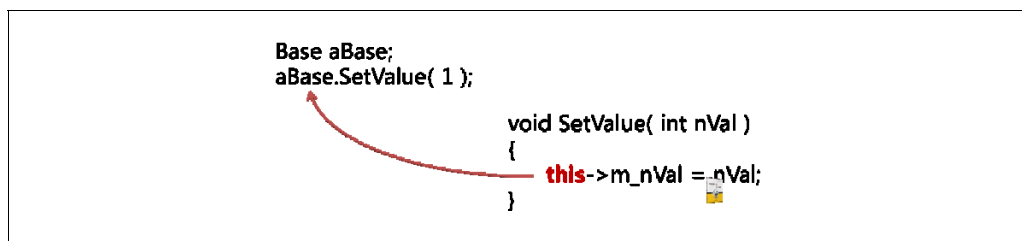


图 6-17 this 指针

· C++世界的奇人异事

在武侠小说中，初入武林的毛头小子总是要遇到几位奇人，发生几件异事。经过高人的指点，经历一番磨练，方能武功精进，从一个新手成长为高手。在 C++ 世界，同样有诸多的奇人异事。在 C++ 世界中游历学习的我们，是否也期望遇到几位奇人，经历几件异事，而从一个 C++ 新手成长为 C++ 高手呢？

武林中的奇人异事可遇而不可求，但是 C++ 世界中的奇人异事却可以由我为你一一引见。



7.1.1 指针的运算

这里大家肯定会奇怪，我们对指针进行的是加 1 的运算，怎么指针指向的地址却增加了 4 个单位？这是因为指针跟它所指向的数据的真正数据类型相关，指针加 1 或者减 1，会使指针指向的地址增加或者减少一个对应的数据类型的长度。比如以上代码中的 int 型指针，它的加 1 运算就使地址增加了 4 个单位，也就是一个整型数的长度。同理，对字符型指针加 1，地址实际增加 1；对双精度型指针加 2，地址实际增加 16。指针偏转流程如图 7-1 所示。

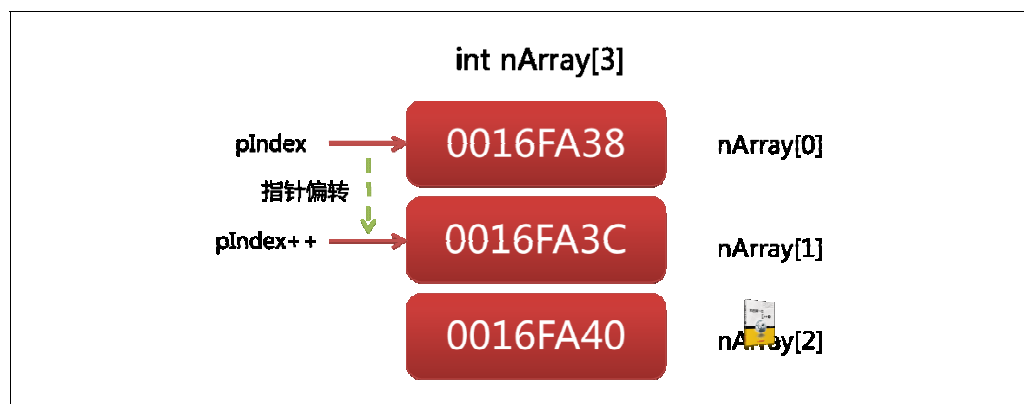


图 7-1 指针偏转

7.1.3 指向指针的指针

指针变量可以指向整型变量、字符型变量等基本数据类型的变量，也可以指向指针类型变量。当指针变量用于指向指针类型变量时，称为指向指针的指针变量。这句话虽然有些像绕口令，但其实可以这样理解：指针也是一个变量，在内存中的某个地址存放着这个变量，当有另外指针指向这个地址时，这个指针就是我们所说的指向指针的指针了。怎么？还是难以理解？没有关系，下面来看一个实际的例子：

```
int N = 2;
int* pN = &N;
int** ppN = &pN;
```

在这段代码中，首先定义了一个整型变量 N，然后定义了一个整型指针指向这个变量 N。换句话说，这个指针的值就是整型变量 N 在内存中的地址。然后，指向指针的指针登场了，我们用“**”定义了一个指向 pN 指针的指针，ppN 中保存的就是 pN 指针变量的地址，也就是它指向这个整型指针。图 7-2 展示了这三个变量之间的内存关系。

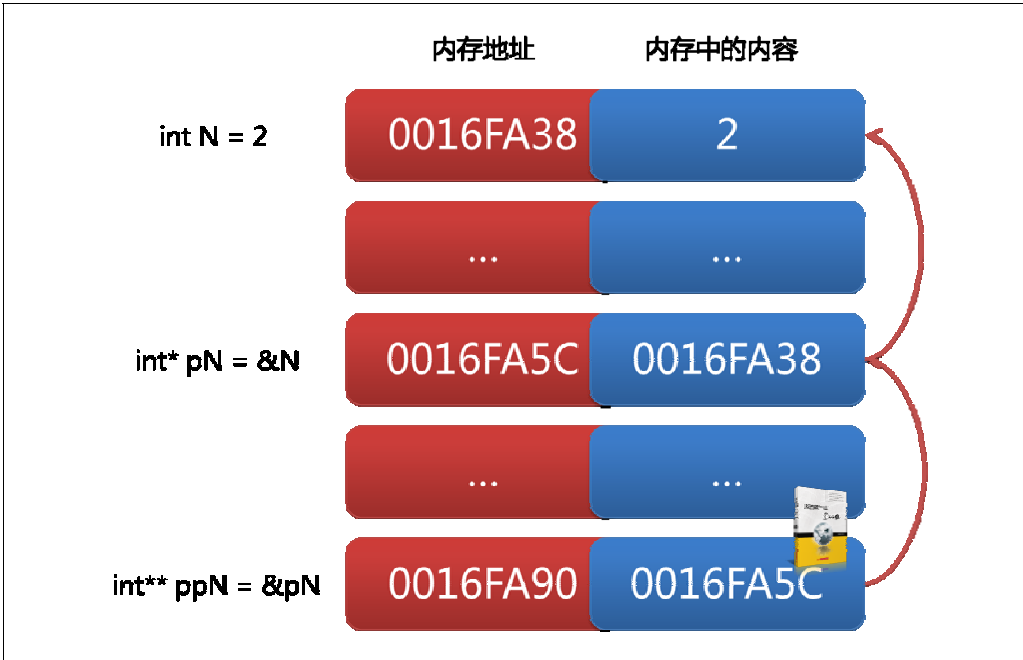


图 7-2 指向指针的指针

7.1.4 指针在函数中的应用

在主函数中，我们调用了数组求和函数 SumArray()，并将数组的起始地址 nArray 和保存结果的整型变量的地址 &nArraySum 传递给了求和函数。在求和函数 SumArray() 中，通过传入的数组地址访问整个数组，完成传入数据的功能。同时，利用指向保存结果变量的指针，将结果直接保存到变量 nArraySum 中，完成传出数据的功能。利用指针作为函数参数传递数据的本质，就是在主调函数和被调函数中，通过指向同一内存地址的不同指针访问相同的内存区域，从而实现数据的传递和交换。图 7-3 展示了指针作为函数参数访问相同内存的结果。

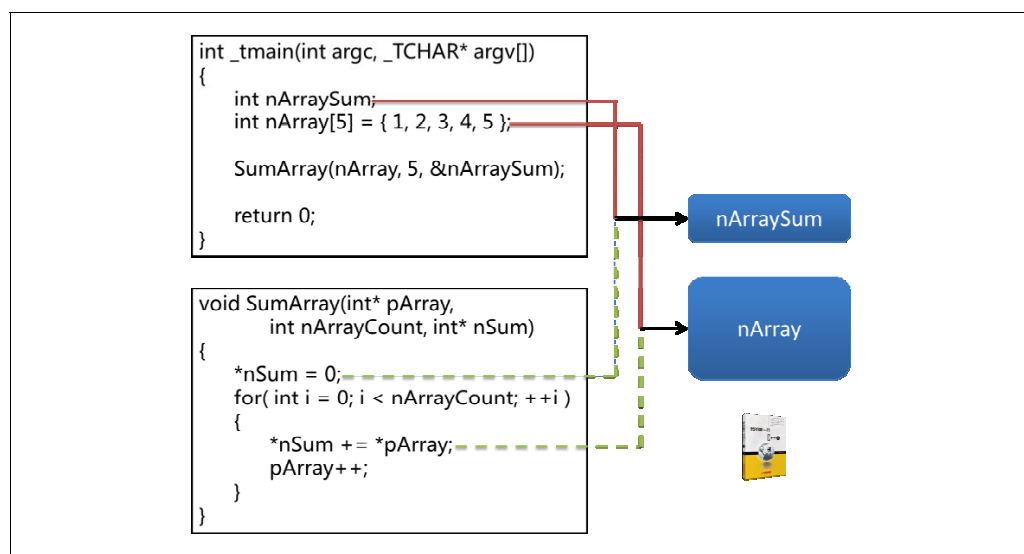


图 7-3 指针作为函数参数访问相同内存

从程序的输出结果可以分析这三种传递参数方式的区别：在函数 `FuncByValue()` 中，其内部的形式参数 `x` 只是外部实际参数 `n` 的一份拷贝，当对 `x` 进行运算时不能改变 `n` 的值，所以输出结果仍然为 0；在函数 `FuncByPointer()` 中，指针 `p` 是指向外部变量 `n` 的指针，改变指针 `p` 所指向的数据的值实际上就是改变 `n` 的值，所以在函数内部对指针 `p` 所指向的数据的改变，就直接反映到 `n` 的数值的修改，输出结果为 1；在函数 `FuncByReference()` 中，引用 `r` 就是外部变量 `n` 的引用，引用 `r` 和变量 `n` 都是同一个数据，在函数内部改变 `r` 同样也会修改 `n`，所以最终输出结果为 2。对比以上三种传递参数的方式可以发现：传引用的性质跟传指针相似，既可以传入参数也可以传出参数；同时传引用又跟传值相似的书写形式，它可以直接使用变量调用函数，在函数内部引用的使用方式也跟普通变量的使用方式一样。这样传引用既享受了传指针的好处——节省空间，提高效率，可以用作参数的传入和传出，又跟传值保持了相同的书写形式——让引用在函数中使用起来更加简单自然，代码的可读性也更高。可以说，传引用同时具备了传指针和传引用两者的优点。所以，在可以的情况下，都应该尽量使用传引用来传递函数参数，而尽量减少使用传指针（指针使用具有一定的危险性）和传值（效率低下）这两种形式。简单来讲，就是尽可能地使用引用，不得已时才使用指针。三种传递参数的方式如图 7-4 所示。

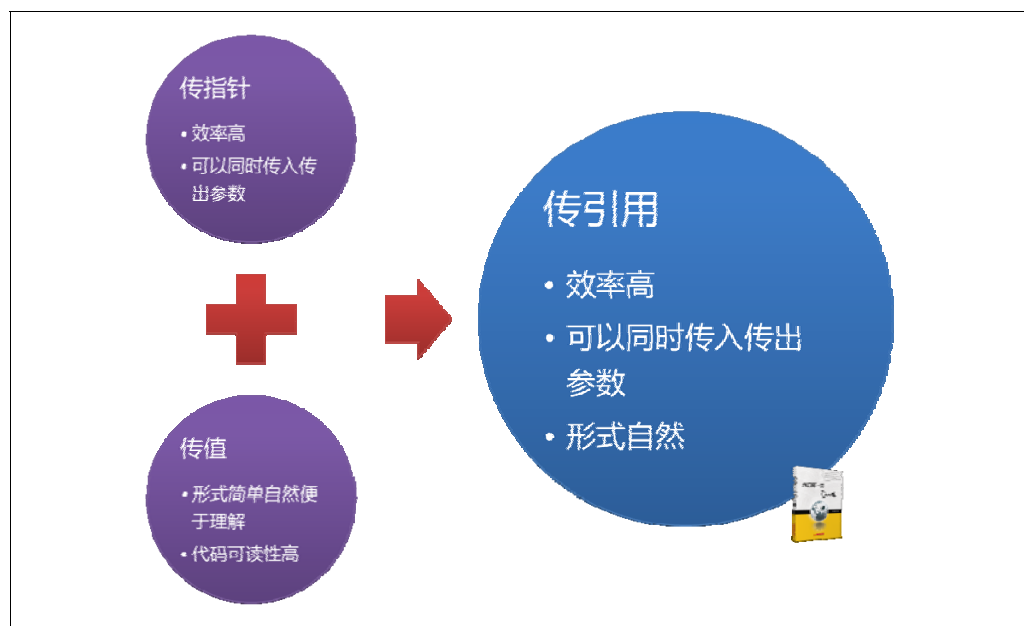


图 7-4 优先选择传引用

7.2.1 异常处理

在这段程序中，我们在 try 语句块中调用了 Divide() 函数，而 Divide() 函数在执行过程中会对除数进行检测，当检测到除数为 0 时，它就用 throw 关键字抛出一个描述了错误信息的字符串异常。在 try 语句块之后的 catch 语句会捕获到这个字符串异常并对其进行处理，这里对异常的处理只是将这个错误信息输出报告给用户。在实际应用中，异常处理往往要比这复杂，比如要进行资源的清理、记录错误日志等。异常处理的三个步骤如图 7-5 所示。

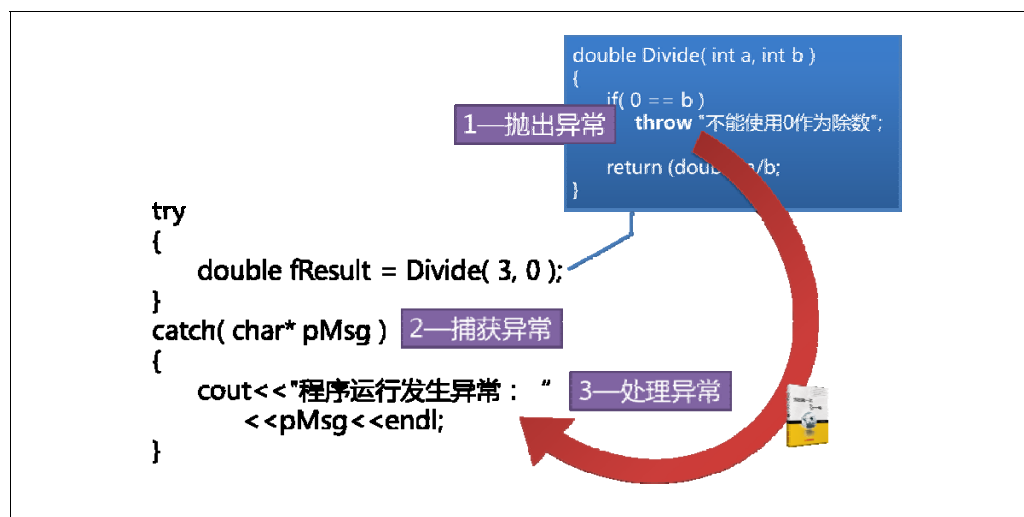


图 7-5 异常处理三部曲

7.3.1 源文件和头文件

4. 从预处理的角度讲

对于包含了多个头文件的源文件，只要用头文件的内容替换掉源文件中对应的#include语句，就可以得到预处理后的源文件。这种经过预处理后生成的源文件的编译结果就是最终的编译结果。换句话说，头文件可以看成是多个源文件的共有部分。这些共有部分被抽取出来成为一个头文件。反过来，这个头文件又被多个源文件引用，以此来达到代码复用的目的。源文件和头文件的包含关系可以用图 7-6 表示。

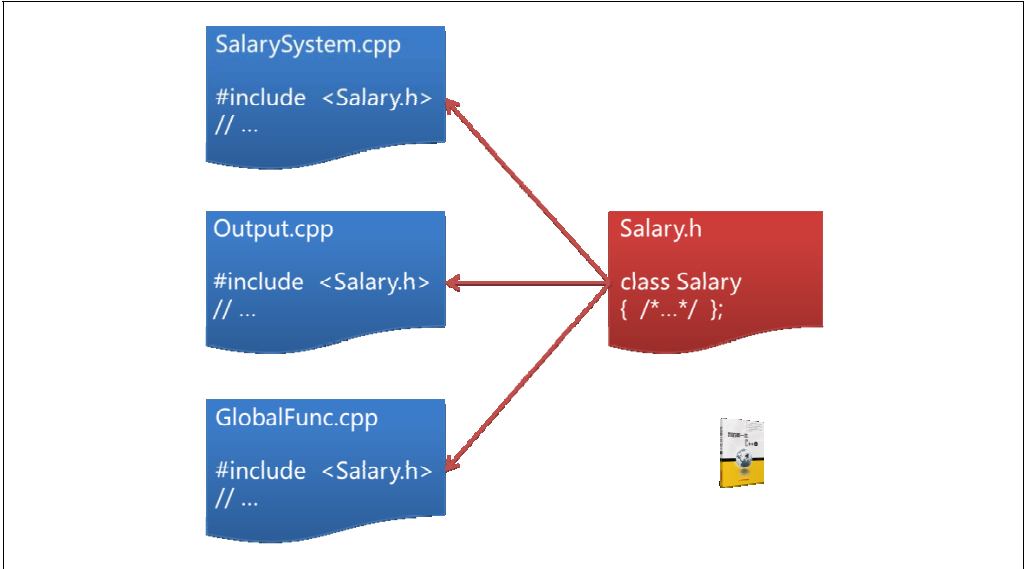


图 7-6 源文件和头文件的包含关系

7.3.2 名字空间

在一些大型系统中，因为多人参与共同开发，所以即使采用了源文件、头文件分离的方式，也难以保证函数、数据声明的唯一性。比如，程序员张三声明了一个名为 Student 的数据类型，而李四在不知情的情况下，同样声明了另外一个名为 Student 的数据类型，这样在同一个系统中的两个不同的 Student 就会产生冲突。为了解决这个问题，C++提供了名字空间（namespace）来规划和管理程序结构。

从以上代码中可以看到，我们在名字空间 Zhangsan、名字空间 Lishi 和全局名字空间中分别定义了 Student 结构体，并且 Student 结构体的实现各不相同，如图 7-7 所示。这是因为 Student 结构体分别属于不同的名字空间，所以虽然这些结构体的名字相同，但是并不会产生冲突，就像你家有电视机，我家也有电视机，但实际上却是两个不同的电视机，互不影响。

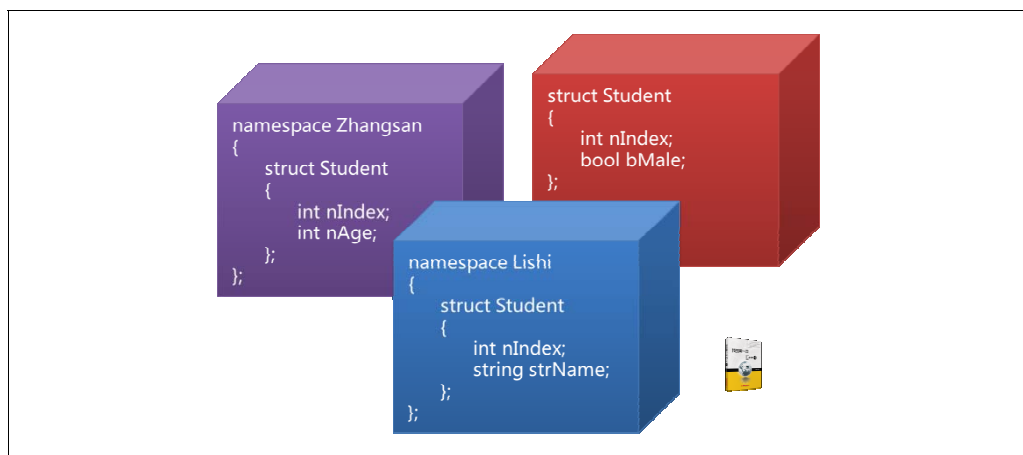


图 7-7 名字空间的包装功能

7.3.3 作用域与可见性

在这个例子中，我们在函数体作用域的开始定义了变量 `nNum`，那么它的作用域就是其定义后从开始到函数最后结束。而第二个变量 `nNum` 则定义在一个局部作用域中，那么它的作用域是其定义后从开始到“`{}`”语句块最后结束。虽然是相同的变量名，但是因为处在不同的作用域中，所以不会引起冲突，同时又因为子作用域的优先级高于父作用域，如“`nNum = 1`”赋值语句实际上是对第二个局部作用域中的变量 `nNum` 进行赋值，不会影响父作用域中的 `nNum` 变量，所以最后输出函数作用域内的变量 `nNum` 的值还是最开始的初始值。虽然这样的做法在 C++语法上是允许的，但是为了避免代码在语义上的混淆，最好不要采用这样的方式在一个函数中定义两个同名的变量，如图 7-8 所示。

```
void foo(void)
{
    int nNum = 0;

    {
        int nNum;
        nNum = 1;
        cout<<"在局部作用域中输出： "
            <<nNum<<endl;
    }

    cout<<"在函数体作用域中输出
        <<nNum<<endl;
}
```

图 7-8 作用域与局部作用域

2. 全局作用域

// Global.cpp: 定义全局变量和全局函数

```
#include "StdAfx.h"
// 全局变量
int gTotal = 0;
// 全局函数
int Add( int a, int b )
{
    return a + b;
}
```

如果想在另外一个源文件中使用这个全局变量和全局函数，就需要用“extern”关键字对它们重新进行声明，然后才可以开始使用。

```
// HelloWorld.cpp: 使用全局变量和全局函数
// 在变量声明前加上“extern”关键字，重新声明全局变量
extern int gTotal;
// 在函数声明前加上“extern”关键字，重新声明全局函数
extern int Add( int a, int b );

int _tmain(int argc, _TCHAR* argv[])
{
    // 使用全局变量和全局函数
    gTotal = Add(2,3);

    return 0;
}
```

这样，通过“extern”关键字，变量或者函数就可以冲出单个源文件，走向整个项目。我们可以在多个源文件之间共享全局变量和全局函数，实现真正的“全局”作用域，如图 7-9 所示。

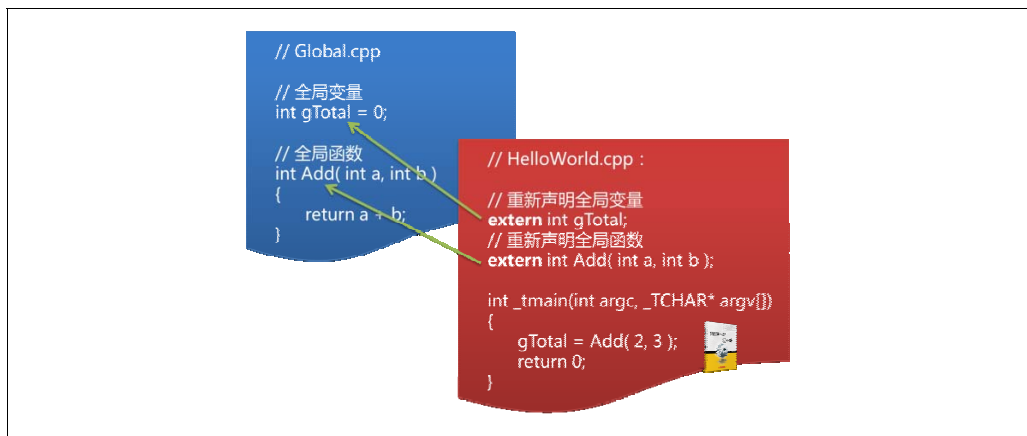


图 7-9 多个源文件共享全局变量和全局函数

7.4.1 偷梁换柱：宏定义

宏定义又称为宏替换，简称“宏”。在 C++ 中，我们使用“#define”指令来定义一个宏：

```
#define 标识符字符串
```

其中，标识符就是所谓的符号常量，也称为“宏名”。当定义好宏之后，就可以在程序代码中使用这个宏来代替宏定义中的字符串。例如：

```
// 定义一个宏
#define PI 3.14159

// 利用定义的宏计算圆面积
double fR = 5.0f;
double fArea = PI * fR * fR;
```

当预编译程序对程序代码进行预处理时，如果遇到代码中使用了宏，就会将宏展开，也就是将宏名替换为宏定义中的字符串。所以，宏展开后，这段代码实际上成为

```
// 宏展开后的代码
double fArea = 3.14159 * fR * fR;
```

这里我们可以看到，宏的本质就是“替换”，也就是偷梁换柱：偷去程序代码中的宏名，换上宏定义中的字符串。虽然在程序代码中使用的是宏，但是经过预处理后最终参与编译的代码却是“替换”后的代码。所以，要理解别人使用宏的代码，先要进行“替换”，同样，如果代码中有需要进行“替换”的地方，比如这些长长的字符串常量可以替换为一个简短的宏，或者利用宏给某个字符串一个更有意义的名字，都可以使用宏，如图 7-10 所示。

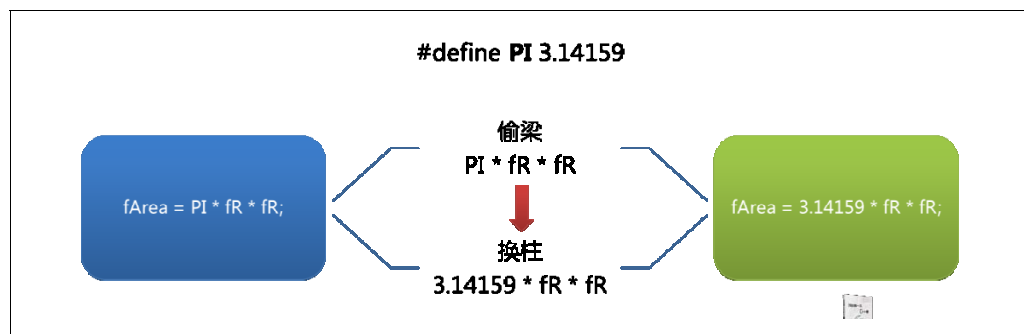


图 7-10 宏的偷梁换柱

！

7.4.3 固若金汤：使用 const 保护数据

2. 使用 const 表示常量

使用 const 关键字可以使编译器检测出对常量的非法操作，以防止常量被修改。const 关键字常见的常量声明方式如下：

```
//声明一个整型常量并赋初值为 1
const int number = 1;
// 声明一个 Teacher 类型的常量对象
const Teacher MrChen;
//声明一个常量整型指针，指针所指向的变量的值不能修改，也就是一个常量
const int* pNumber;
//声明一个常量整型指针，意义同上
int const * pNumber;
//声明一个整型常量指针，指针不能修改
int* const pNumber = &number;
//声明一个常量整型常量指针
// 指针和指针所指向的变量值都不能修改
const int * const pNumber = &number;
//声明一个常量整型引用
const int & number = number;
```

这里需要特别注意的是，第二种常量整型指针和第四种整型常量指针的区别。因为 const 关键字位置的不同，导致这两种形式表达的意义差别很大。为了弄清楚这两种形式的差别，我们可以以 “*” 为界，把声明语句分割成两个部分，如图 7-11 所示。

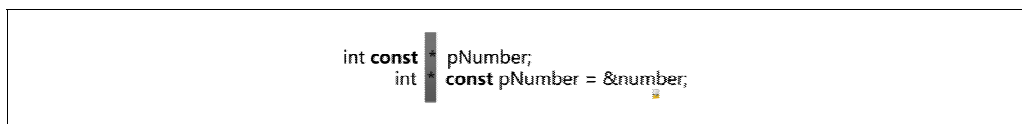


图 7-11 使用*分割常量整型指针和整型常量指针

第三篇用 STL 优雅你的程序



C++世界的优雅绅士：STL 标准模板库

在见识了 C++世界的奇人异事之后，我们自然是功力倍增，信心满满，遇到什么事情都想用 C++来解决，以期在众人面前显摆自己的功力是如何如何高深。正好，这天老板对我们前面完成的工资统计程序提出了新需求：统计员工中拿高工资的人数，也就是工资高于 1 000 元的人数。看到这个新需求，我们一定会在心中暗暗发笑：这还不简单，小菜一碟啊！很快，就有了自己的实现方法。

```
// 统计高工资人数
const int MAX_COUNT = 10000;
int _tmain(int argc, _TCHAR* argv[])
{
    // 定义保存员工信息的数组
    int arrSalary[MAX_COUNT];
    // 当前员工个数
    int nCount = 0;
    // 用户输入...

    // 高工资的员工数
    int nTotal = 0;
    // 循环遍历数组进行统计
    for( int i = 0; i < nCount; ++i )
    {
        if( arrSalary[i] > 1000 )
            ++nTotal;
    }
    // 结果输出...
}
```

正当我们暗自得意、想要把程序拿到老板面前去邀功请赏时，这时过来一位白衣飘飘的绅士，对我们的代码看了两眼，嘴里说出一句话来：

“粗鲁！四肢发达的程序员才这么干！”

自己非常满意的代码居然被人说成是“粗鲁”的代码，心中难免有些不爽，转头想找他理论理论，没想到他已经飘然走远了。为了以后好找他算账，于是赶紧问：

“请问大侠大名？”

“标准模板库，你也可以叫我 STL。”

听到这里，我们不由得暗暗叹道：原来他就是传说中的优雅绅士 STL 啊，今日得见，果然名不虚传、气度非凡啊。心中害怕错过这个结识的机会，于是连忙说道：

“STL 大侠，别走啊，我们交个朋友吧！”



8.1.1 算法 + 容器 + 迭代器 = STL

STL (standard template library, STL), 即标准模板库, 是一个具有工业强度的、高效的 C++ 程序库。据 C++ 世界的老人们说, STL 最初诞生于惠普实验室, 它是由 Alexander Stepanov、Meng Lee 和 David R. Musser 在惠普实验室工作时所开发出来的, 后来经过不断地发展形成不同的版本。现在, 它已经被容纳于 C++ 标准程序库 (C++ standard library) 中, 是 ANSI/ISO C++ 标准中最新的也是极具创新性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法, 为广大 C++ 程序员提供了一个可扩展的应用框架, 高度体现了软件的可复用性。我们不仅可以直接从中获得极大的开发便利, 也可以通过继承现有类, 自己编制符合接口规范的容器、算法、迭代器等方式对之进行扩展。

从广义上讲, STL 主要分成三大核心部分: 算法 (algorithm)、容器 (container) 和迭代器 (iterator), 除此之外还有容器适配器 (container adaptor)、函数对象 (functor) 等。几乎 STL 的所有代码都采用了模板类和模板函数的方式, 这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。下面我们先对 STL 的三大核心部分做一个简单了解。

正是算法、容器和迭代器这三个核心部分, 它们相互配合, 相互作用, 使得 STL 成为了一个有机的整体, 如图 8-1 所示。

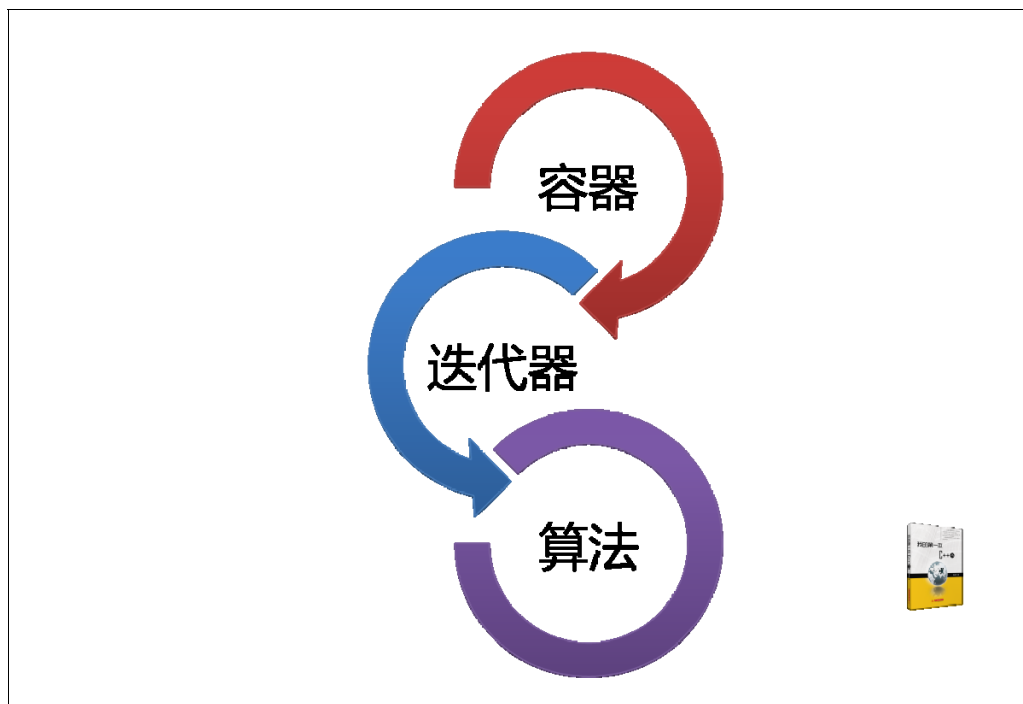


图 8-1 STL 的三大核心部分

8.2.1 函数模板

实际上，我们可以把“typename 标识符”看成是函数模板的参数，不同参数可以产生不同版本的函数。编译器在编译 mymax()函数调用时，可以根据参数的类型推导出函数模板的这个类型参数。例如代码中第一次调用 mymax()函数，其参数都是整数，则编译器推导出其模板参数 T 为 int，此时，编译器就会以函数模板为样板，也就是用实际的数据类型替换函数模板中的类型参数 T，自动为这个函数调用生成一个整型数的版本。

```
// 整型数版本的 mymax() 模板函数
int mymax( int a, int b )
{
    return a > b ? a : b ;
}
```

当主函数以整型数为参数调用 mymax()函数时，实际上执行的是上面生成的整型数版本的 mymax()函数。同理，当我们以浮点数为参数调用 mymax()函数时，编译器也会为这个 mymax()函数调用生成一个浮点数版本的 mymax()函数。函数模板生成重载函数可用图 8-2 所示。

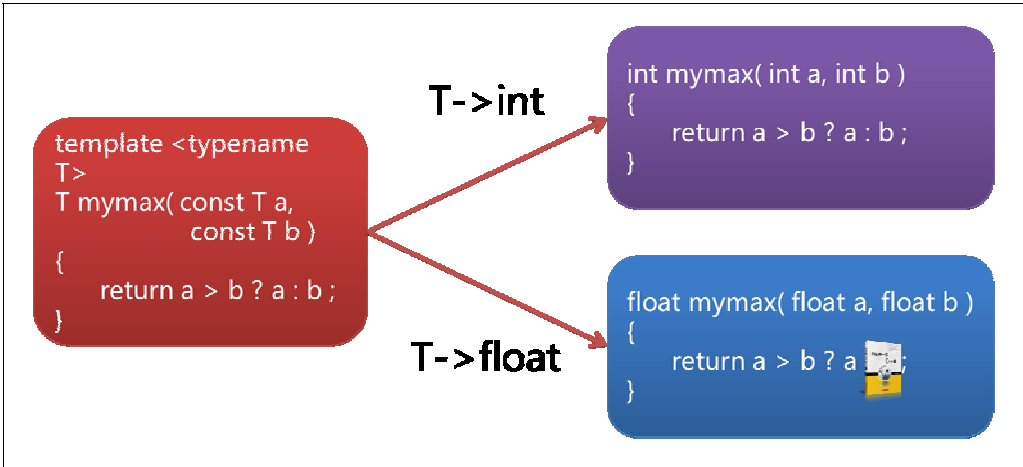


图 8-2 函数模板生成重载函数

8.2.2 类模板

```
int _tmain(int argc, _TCHAR* argv[])
{
    // 用 int 数据类型对类模板进行实例化
    // 比较两个整数的大小
    compare<int> intcompare(2,3);
    wcout<<intcompare.max()<<" 大于"<<intcompare.min()<<endl;

    // 用 string 数据类型对类模板进行实例化
    // 比较两个字符串的大小
    compare<string> stringcompare("A","B");
    cout<<stringcompare.max()<<" 大于"<<stringcompare.min()<<endl;

    return 0;
}
```

在主函数中，我们首先使用 `int` 数据类型实例化了一个 `compare<T>` 类模板，形成了一个新的模板类 `compare<int>`，然后定义了一个 `compare<int>` 类的对象 `intcompare`，用于比较两个整数。同样，我们使用 `string` 数据类型实例化 `compare<T>` 类模板可以得到模板类 `compare<string>`，这个类可以用于比较两个字符串的大小。从类模板到模板类实例的过程可用图 8-3 表示。

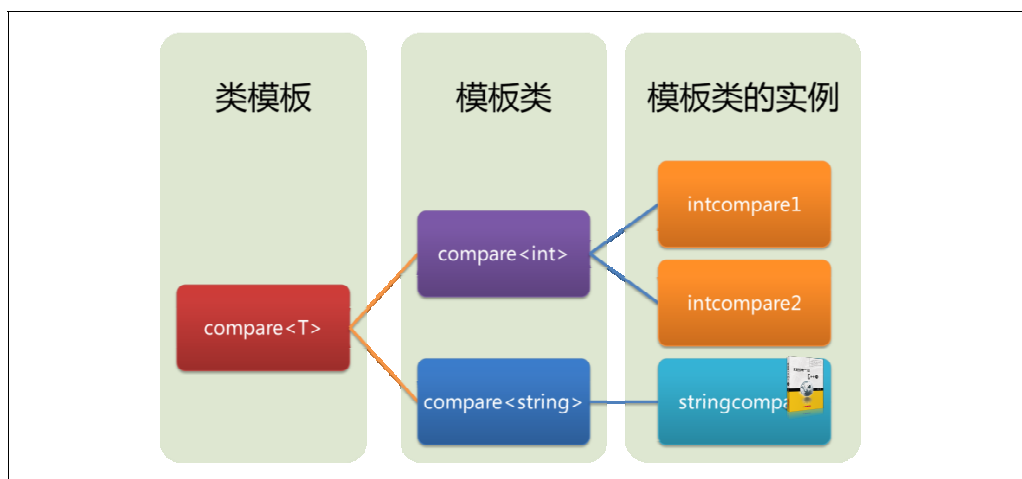


图 8-3 从类模板到模板类实例的过程

大肚能容：STL 中的容器

虽然 STL 中有算法、容器和迭代器三大核心组件，但是最受 C++ 程序员欢迎的还是大肚能容的容器。

我们知道，程序就是用来处理数据的。一个程序从它的开始执行到执行完毕，无时无刻不在跟各种数据打交道。在一个程序中，对于少量的数据，我们使用单个变量就可以搞定。但如果要处理大量的数据，就需要对这些数据进行良好的管理。正所谓忆苦才能思甜，在容器诞生之前，程序员们在保存和管理大量数据时使用最多的是数组，这是程序员们刀耕火种的年代。使用数组，程序员们需要自己做很多事情，比如管理内存，维护数组中保存的数据、防止数组访问越界等。随着 STL 容器的到来，程序员们纷纷扔掉镰刀锄头，开上了既省时又省力的拖拉机。相比于数组，容器更加强大而灵活：它们还可以……

关于容器的好处说不完，这也就不难理解它们为何如此受到程序员的欢迎了。STL 容器不是一般地好，而是确实很好。



STL 中的容器分为顺序容器（sequence container）和关联容器（associative container）两种。STL 中的容器可以用图 9-1 来描述。

1. 顺序容器

顺序容器将对象组织成有限线性集合，所有对象都是同一类型。STL 中包括三种基本顺序容器：向量（vector）、线性表（list）和双向队列（deque）。基于这三种基本顺序容器，又可以构造出一些专门的容器，用于比较特殊的数据结构，包括堆（heap）、栈（stack）、（queue）、队列（queue）及优先队列（priority queue）等。

2. 关联容器

关联容器所容纳的对象是由{键，值}对组成的，它提供了基于键值的数据快速检索能力。容器中的元素在加入容器时就已经被排好序，检索数据时可以按照二分搜索提高检索的效率。STL 中有八种关联容器。当一个键对应一个值时，可以使用集合（set）和映射（map）存放这种一一对应的数据。若同一个键对应多个值时，则可以使用多集合（multiset）和多映射（multimap）存放这种一对多的数据。同时，集合和映射又可以根据内部实现机制的不同，分为基于红黑树实现的 set、multiset、map 和 multimap，以及基于哈希表实现的 unordered_set、unordered_multiset、unordered_map 和 unordered_multimap。在使用的时候，我们可以根据需要灵活地进行选择。

图 9-1 STL 中的容器

9.1.2 使用迭代器访问容器中的数据元素

```
// 定义一个 vector<int>容器的迭代器，表示起始位置  
  
vector<int>::iterator itfrom;  
// 将迭代器指向 vector 容器的起始位置 itfrom = vecSalary.begin();  
// 定义一个 vector<int>容器的迭代器，表示终止位置  
vector<int>::iterator itto;  
// 将表示终止位置的迭代器指向 vector 容器起始位置后的第四个数据元素  
itto = vecSalary.begin() + 3;
```

在这段代码中，我们定义了两个 vector 容器的迭代器，分别指向 vector 容器中的第一个元素和第三个元素，以此来表示一个数据元素的范围。使用迭代器表示容器中的范围可用图 9-2 表述。

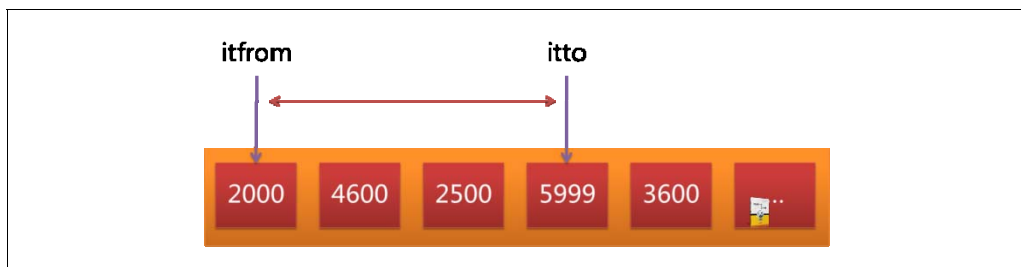


图 9-2 使用迭代器表示容器中的范围

9.2.1 创建并初始化 vector 对象

要使用一个 vector 容器，我们首先需要创建相应的 vector 容器对象。跟所有容器对象的使用一样，我们需要使用容器保存的对象类型实例化一个 vector 容器模板类，然后利用这个模板类创建相应的容器对象实例。

```
#include <vector>           // 引入 vector 所在的头文件
using namespace std;        // 使用 vector 所在的名字空间

// ...
// 使用 Employee 类实例化 vector 类模板，
// 然后创建实例对象 vecEmployee
vector<Employee> vecEmployee;
```

这样，就得到了一个空的 vector 容器，如图 9-3 所示。如果想得到一个已经保存有默认数据的 vector 容器，则可以在创建容器对象的时候，利用其构造函数指定默认数据及默认数据的个数。当容器对象创建完成时，这些默认数据也就已经保存在容器中了。例如：

```
Employee employee;
// 创建 4 个 employee 对象的副本保存到容器中
vector<Employee> vecEmployee(4, employee);
// 使用 Employee 类的默认构造函数创建 4 个对象保存到容器中
vector<Employee> vecDefEmployee(4);
```

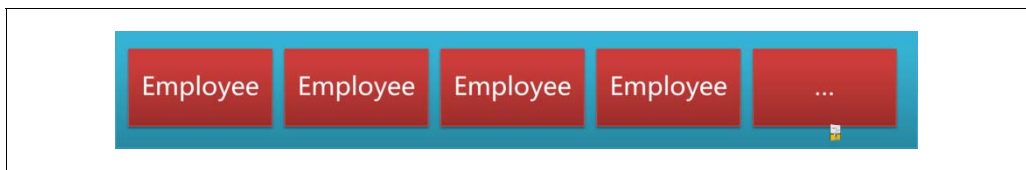


图 9-3 vector 容器

9.3.1 创建并初始化 map 容器

map 是 STL 中的一种关联容器，它提供了一种对数据对进行保存和管理的能力。这种数据对是一对一成对出现的，其中，第一个数据可以称为数据对的键（key），且每个键只能在 map 中出现一次；而第二个数据可以称为该键的值（value）。例如，一个员工号和一个员工对象共同构成一个数据对，员工号是这个数据对的键，而员工对象就是这个数据对的值。在 map 容器的内部，它是使用一颗红黑树实现的——一种非严格意义上的平衡二叉树，如图 9-4 所示。这颗树具有对数据自动排序的功能，所以 map 内部所有的数据都是有序的。正是这种特性，使得 map 容器在增加和删除节点时对迭代器的影响很小，除了被操作的当前操作节点，对其他节点都没有什么影响。所以 map 容器特别适用于对大量数据进行保存和管理。

跟 vector 容器的使用相似，map 容器也同样是一个类模板，我们在使用的时候，需要给定的数据类型参数对其进行实例化以形成能够保存具体数据类型数据的模板类。因为管理的是数据对，所以 map 类模板需要两个数据类型作为模板参数，其中第一个是键的数据类型，而第二个就是对应的值的数据类型。例如：

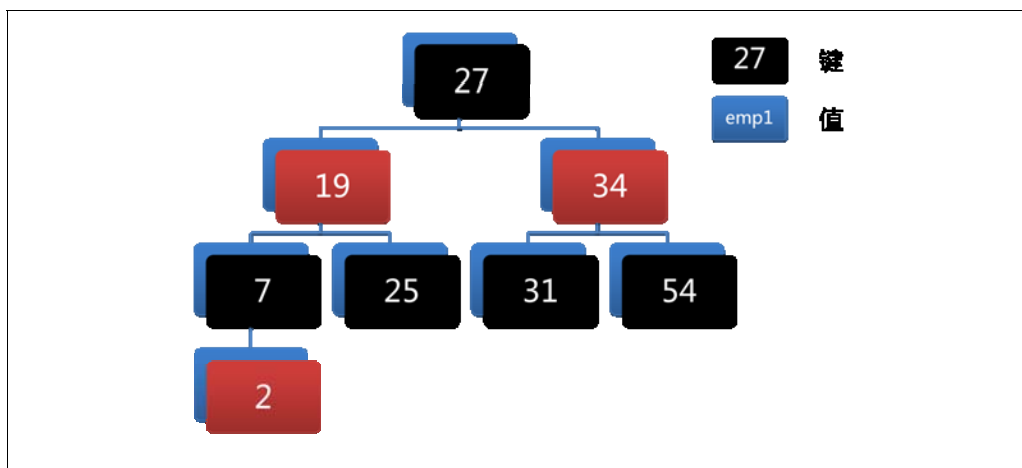


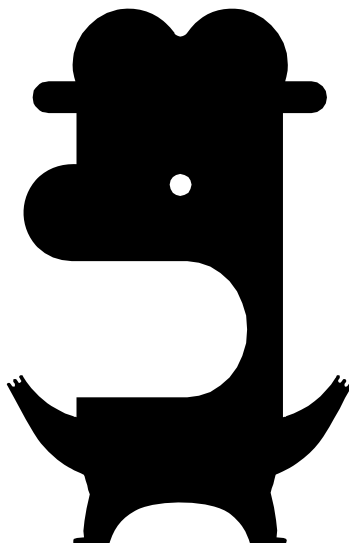
图 9-4 红黑树实现的 map 容器

通吃天下：STL 中的通用算法

程序的两大核心任务就是管理数据和处理数据。使用 STL 中的各种容器，可以对各种数据进行良好的管理，但是只管理好数据是没有用的，更重要的是还要对数据进行处理，得到最终的结果，而这正是 STL 通用算法的用武之地。STL 提供了大量的常用算法，利用这些算法可以轻松完成对数据的常见处理，比如数据的查找、排序、填充等，并且算法的效率往往比自己动手实现的相同算法的效率要高。

更重要的是，STL 库中的这些算法都以迭代器类型为参数，这就和数据所在容器的具体实现相互分离了，我们可以把一个排序算法用于 vector 容器，也同样可以将这个排序算法用于 map 容器。这样，就实现了算法的大小通吃——不管数据保存在什么容器，也不管容器中的数据到底是什么数据类型，算法都能够处理。这也是为什么 STL 中的算法被称作通用算法（generic algorithm）的原因。

既然有这么又简单又高效的东西，我们没有理由不用啊！



10.1 STL 算法中的四大帮派

STL 中的通用算法众多，按照是否改变容器中元素的顺序和是否修改数据元素，算法也各自拉帮结派，形成了 STL 算法中的四大帮派，如图 10-1 所示。

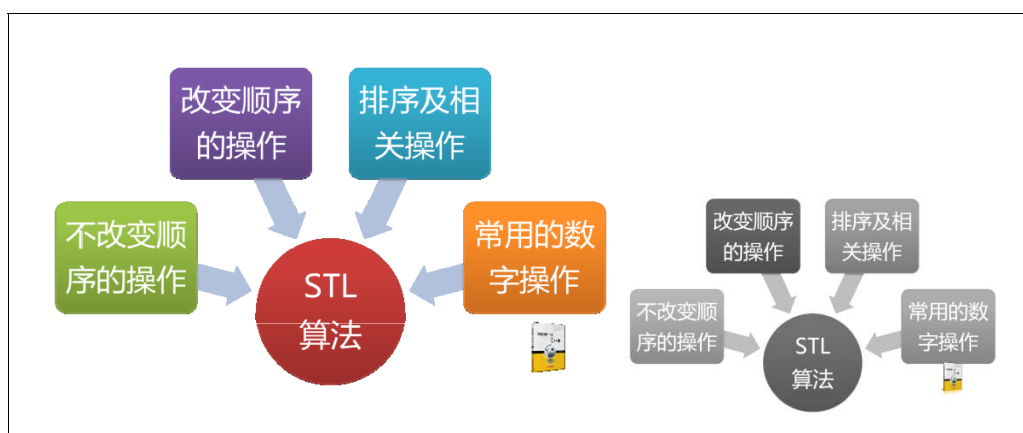


图 1-1 STL 算法的四个帮派

10.2.1 遍历容器中的数据元素：for_each()算法

在这段函数中，我们使用 `for_each()` 算法完成了对容器中所有数据的处理，因为它指定的处理范围是容器的开始位置和末尾位置，当然也可以根据需要改变这个范围，只对某个特定范围内的数据进行处理。`for_each()` 算法在执行过程中，会逐个遍历整个范围内的所有数据元素，并以这些元素为参数调用它的处理函数，也就是用这个函数处理这些元素。在这里，我们将数据元素的处理方法定义在 `addsalary()` 函数中，通过 `addsalary()` 函数的形式参数，`for_each()` 算法将容器中的每个数据逐个传递入 `addsalary()` 函数，然后在 `addsalary()` 函数中对数据进行处理。比如，`addsalary()` 函数会判断工资是否低于 2 000 元，如果低于 2 000 元就增加 30% 的薪水。实际上，这都是对容器中每个数据元素的操作。`for_each()` 算法对数据的处理可用图 10-2 来表述。

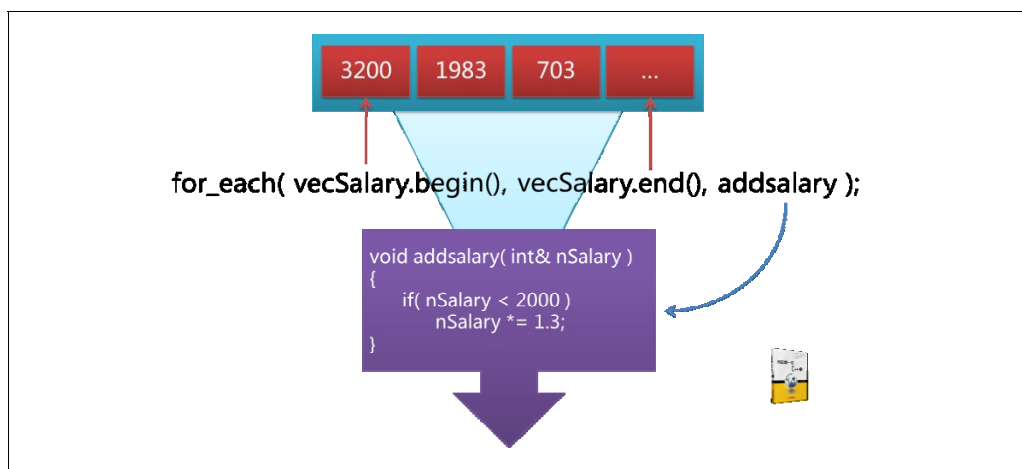


图 10-2 `for_each()` 算法对数据的处理

更形象地讲，`for_each()` 算法更像是一个我们做化学实验时的漏斗装置，我们通过指定容器的范围来决定往这个漏斗中倒入什么数据。而 `for_each()` 算法中的处理函数就是具体的漏斗，它决定什么东西可以留下，什么东西可以漏掉，以此来完成对倒入漏斗中的数据的具体处理。更进一步讲，我们还可以随时更换 `for_each()` 算法中的漏斗，对数据进行不同的处理，这也体现了算法的通用性。

10.3.1 复制容器元素：copy()算法

```
// 保存 C1 和 C2 班级成绩的容器

vector<int> vecScoreC1;
vector<int> vecScoreC2;

// 对容器进行操作，将成绩保存到各自的容器中
// ...

// 保存所有成绩的总容器
vector<int> vecScore;
// 根据各个分容器的大小，重新设定总容器的容量，
// 使它可以容纳即将复制进来的数据
vecScore.resize( vecScoreC1.size() + vecScoreC2.size() );
// 将第一个容器 vecScoreC1 中的数据复制到 vecScore 中
vector<int>::iterator lastit = copy(vecScoreC1.begin(), vecScoreC1.end(),
vecScore.begin() );
// 将第二个容器 vecScoreC2 中的数据追加到 vecScore 的末尾
copy(vecScoreC2.begin(), vecScoreC2.end(), lastit );
```

在以上这段代码中，首先，我们分别使用了两个容器来保存两个班级的成绩，然后又定义了另外一个容器 `vecScore` 来保存两个班级总的成绩。为了让这个总容器能够容纳所有的成绩，我们使用 `resize()` 函数对 `vecScore` 容器的容量作了调整。接着，利用 `copy()` 函数将 `vecScoreC1` 中的所有数据从开始位置到末尾位置，复制到 `vecScore` 容器的开始位置。`copy()` 函数复制完成后，它会返回一个目标容器的迭代器，它指向的是所有复制进来的数据的末尾位置，而这个迭代器，正是我们第二次复制的目标位置。利用两次复制，可以成功地将两个容器中的数据复制到新容器中，并实现数据的连接。`copy()` 算法实现的数据连接可以用图 10-3 来表述。

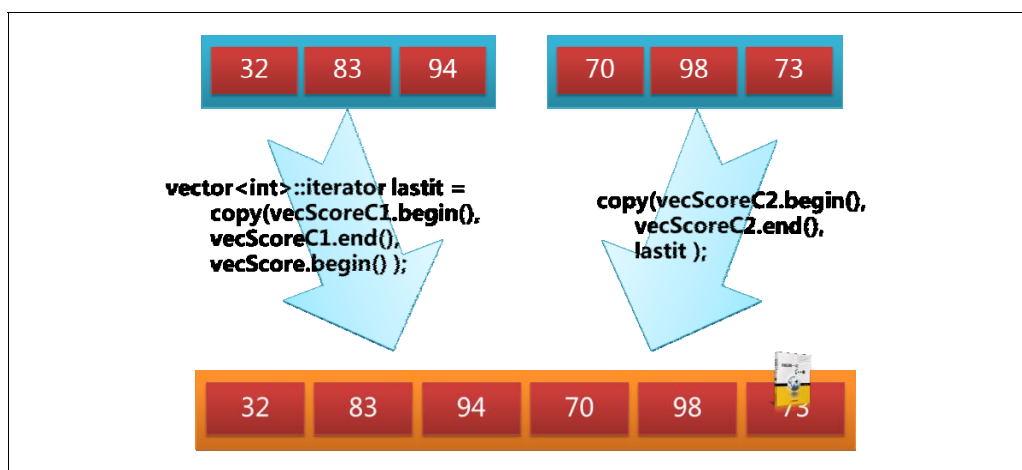


图 10-3 `copy()`算法实现的数据连接

10.4.2 对排序的规则进行自定义

// 定义新的比较函数，实现新的比较规则

```
bool sortbyW( Rect rect1, Rect rect2 )
{
    return rect1.m_fW < rect2.m_fW;
}
```

// 在 sort() 中使用新的比较函数，实现新的比较规则

```
sort( vecRect.begin(), vecRect.end(), sortbyW );
```

在这里，我们很快将新的比较规则实现在比较函数 `sortbyW()` 中，然后通过 在 `sort()` 算法中应用这个比较函数，从而改变 `sort()` 算法原有的比较规则，自然排序的结果也就发生了变化。这正是同一个 `sort()` 算法，不同的比较函数就有不同的比较结果，如图 10-4 所示。

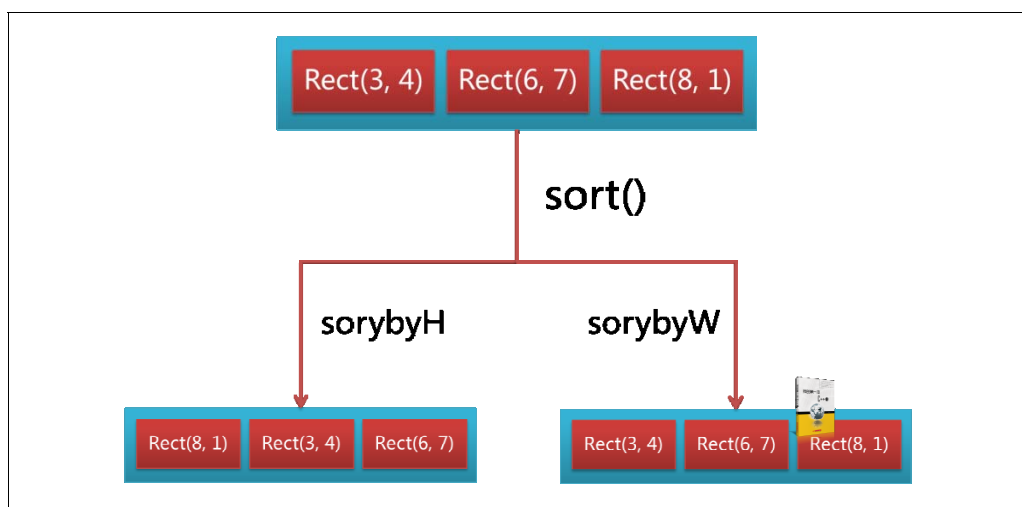


图 10-4 不同的比较函数，不同的比较结果

函数爱穿马甲： 函数指针、函数对象与 Lambda 表达式

在 C++ 世界中，函数的人缘最广，可以说无处不在，到处都有它的身影：从表达运算过程的普通函数到表示类行为的成员函数，特别在 STL 算法中，算法的核心逻辑往往是以函数的形式来表达的。在一些通用算法中，我们通常需要配合一些函数使用，以达到对通用算法进行自定义的效果。比如，一个通用的 `sort()` 排序算法，可以通过向它提供一个排序函数来自定义排序的规则；一个通用的 `find_if()` 查找算法，也可以通过函数对其匹配规则进行自定义。函数极大地增强了通用算法的表达能力，使得通用函数做到了真正通吃天下。

这些能够在通用算法中使用的函数，虽然它们的本质都是函数，但是它们都爱穿马甲，常常变换自己的表现形式：有简单的函数指针，也有复杂的函数对象，更有灵活而优雅的 Lambda 表达式。下面我们就来看看这些函数的马甲，以免它们换个马甲，我们就不认识了。



11.1.3 用函数指针实现回调函数

在这里，我们实际上是通过 PrintMessage()函数定义了一个通用算法框架：首先打印页眉，然后通过函数指针回调函数，打印具体的消息，最后打印页脚。PrintMessage()函数只完成最基本的页眉页脚的打印，至于具体的消息，则留给回调函数负责，这就像留下一个插口，等待某个具体的回调函数插头的插入。在主函数中，通过给 PrintMessage()函数传递不同的打印函数的指针，就如同将某个插头插入 PrintMessage()函数所留下的插口中。不同函数指针插头的插入，可以改变 PrintMessage()函数中的回调函数，进而改变 PrintMessage()函数的行为，达到对其行为进行自定义的效果。回调函数与插头理论的关系可用图 11-1 来描述。

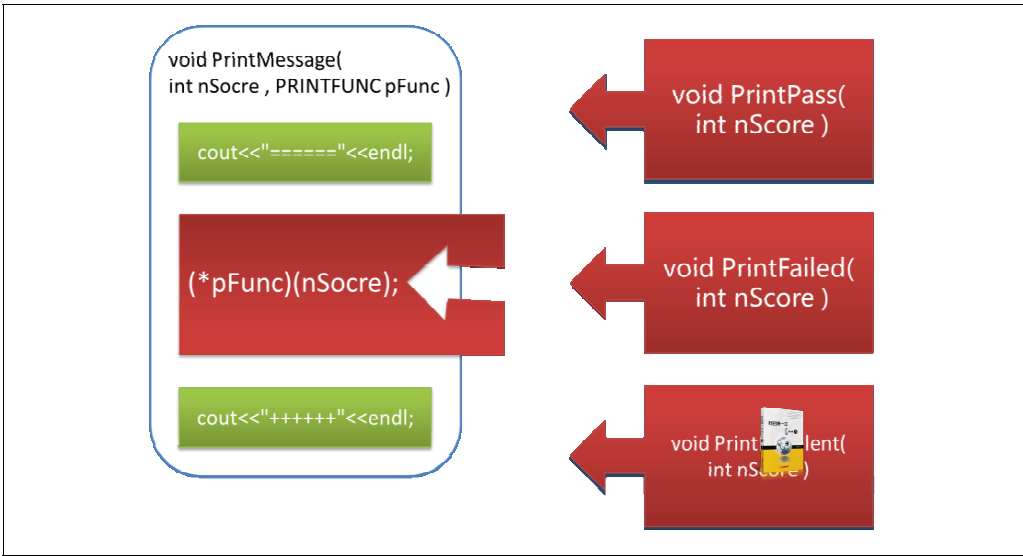


图 11-1 回调函数与插头理论

C++世界的几件新鲜事

“卖报啦卖报啦——”

“C++世界的新标准 C++0x 新鲜出炉，大家都来看看 C++世界的新鲜事啊！”

“右值引用心狠手辣想榨干 C++的性能，智能指针聪明伶俐想解决 C++内存管理的问题，PPL 正在做慈善、大量派发免费午餐啊……”



12.1.2 右值引用在函数返回值上的应用

如果使用右值引用，整个过程就简单多了。首先，我们在函数中创建临时的 MemoryBlock 对象；然后当进行目标对象 block 创建的时候，编译器会检测到对对象进行赋值的是一个右值，这样编译器就直接使用函数返回的指向临时对象这个右值的引用完成 block 对象的创建。这里我们可以看到，block 对象的创建过程，并不是我们通常认为的申请内存、初始化对象属性的过程，而是利用它的右值引用构造函数，直接将整个对象指向右值引用所绑定的对象，即在函数中已经创建的临时对象，再将临时对象移动到目标对象而完成目标对象的创建。通过右值引用，我们将目标对象直接指向一个右值，省略了中间环节中临时对象的创建、销毁及复制过程，实现了右值的废物再利用，低碳又环保，整个过程如图 12-1 所示。

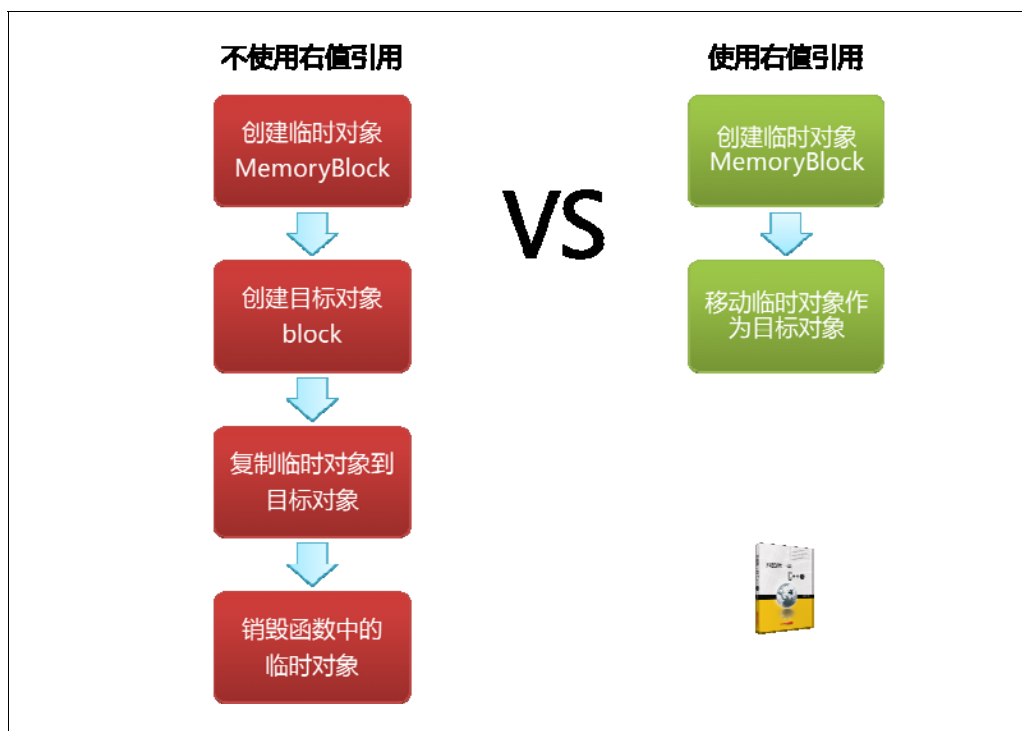


图 12-1 右值引用简化函数返回过程的流程图

12.2.2 用聪明的 shared_ptr 解决内存管理问题

在这段程序中，首先，我们用 new 操作符申请了一块 int 类型的内存，用 pFirst 指针指向这块内存并对其进行管理。这时，指向这块内存的只有 pFirst 指针，所以引用计数为 1。然后，在一个局部作用域中，我们创建了另外一个 pCopy 指针，并用 pFirst 指针对其进行赋值，让它们都指向同一块内存。因为新增了 pCopy 指针指向这块内存，所以引用计数增加为 2。当超出 pCopy 指针的可见域时，pCopy 指针结束其生命周期，这时只有 pFirst 指针指向这块内存，所以引用计数减为 1。当程序最终结束执行返回时，pFirst 指针也结束其生命周期，从此没有任何指针指向此内存资源，引用计数减为 0，内存资源会自动得到释放。整个过程如图 12-2 所示。

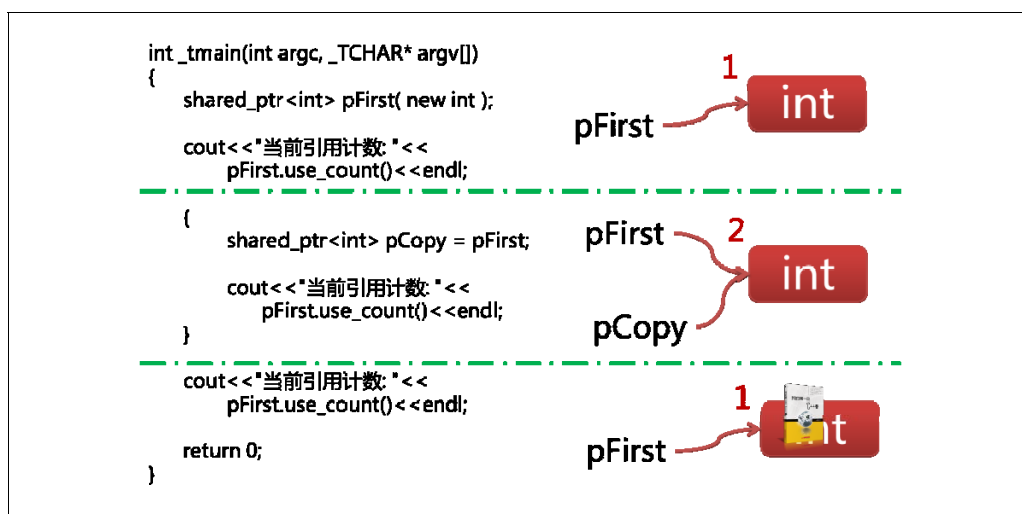


图 12-2 引用计数

12.2.6 对 shared_ptr 自定义

在这段程序中，我们首先定义了一个函数对象 FileClosers，它就是我们自定义的删除器。通过重载它的“()”操作符，在其中调用 fclose()函数关闭文件，完成资源的特殊清理工作。当进行清理工作时，shared_ptr 会将它所管理的指针作为参数传递给删除器对象，所以操作符“()”的参数类型就是它要清理的指针，在这里，也就是 shared_ptr 所管理的 FILE*类型。当创建智能指针 shared_ptr 时，通过指定自定义的删除器，可以让 shared_ptr 在清理资源的时候进行特殊的操作，从而完成 shared_ptr 的自定义。在主函数中，通过 shared_ptr 的构造函数，我们不仅指定 shared_ptr 所管理的指针，还指定其删除器为 FileClosers()，这样 shared_ptr 在释放资源的时候，就会以它所管理的指针作为参数调用这个函数对象，在其中完成特殊的清理工作。整个过程如图 12-3 所示。

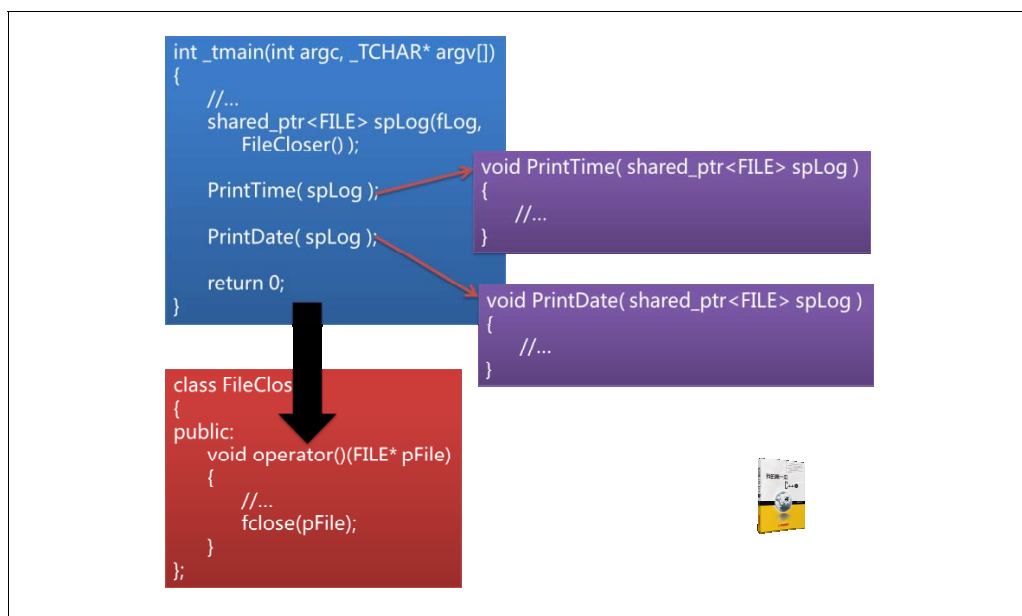


图 12-3 shared_ptr 的使用与销毁过程

12.3.1 都是多核惹的祸：免费的午餐没了

小时候，老师总教育我们上课要专心，“一心不可二用”。可是这个道理在计算机的世界却行不通，因为 CPU 这个不听话的孩子，偏偏在单核 CPU 之外发展出多核 CPU，总想着一芯多用。随着多核 CPU 的推出，一芯多用已成为越来越普遍的事情。从单核到双核，从双核到四核再到八核等，我们开始进入一个一芯多核的时代，如图 12-4 所示。

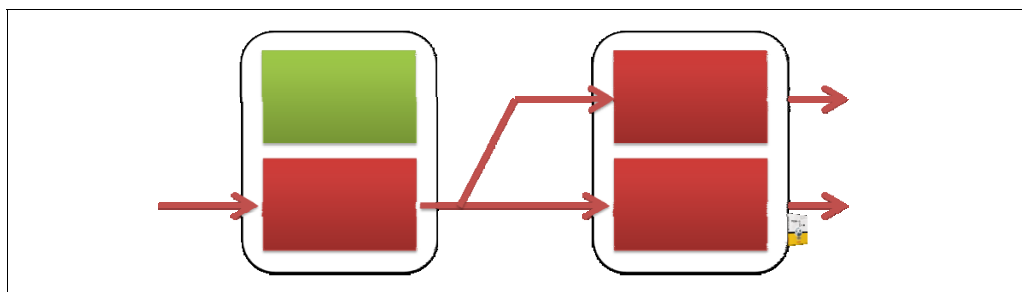


图 12-4 程序在多核 CPU 的串行计算和并行计算

12.3.5 PPL 中的并行对象和并行容器

```
// 保存 Student 对象的 vector 容器
vector<Student> vecStu;

// 将 Student 对象保存到 vector 容器中...

// 使用 combinable<T>模板类创建一个 int 类型的并行对象 ncmbTotalH
combinable<int> ncmbTotalH;
// 使用 parallel_for_each()算法并行地访问 vector 容器,
// 统计 vector 容器中所有 Student 对象的身高总和
parallel_for_each( vecStu.begin(), vecStu.end(),
    [&](Student st)    // 用 Lambda 表达式统计身高总和
    {
        // 将 Student 对象的身高累加到并行对象 ncmbTotalH 的本地副本
        ncmbTotalH.local() += st.GetHeight();
    });

// 合并并行对象 ncmbTotalH 的本地副本, 获得最终结果
int nTotalH = ncmbTotalH.combine(plus<int>());
// 计算并输出平均身高
if( vecStu.size() != 0 )
{
    float fAverageH = nTotalH / vecStu.size();
    cout<<"平均成绩是"<<fAverageH<<endl;
}
```

在这段代码中, `parallel_for_each()` 算法会并行地访问 `vector` 容器中的 `Student` 对象, 然后将每个对象的身高累加到并行对象 `ncmbTotalH` 的本地副本, 最后, 通过并行对象的 `combine()` 函数合并它的多个本地副本就得到了最终的结果。我们可以看到, 各个线程各自独立地访问并行对象的本地副本, 这样就避免了共享资源的互斥与同步, 程序性能自然也会大大提高, 如图 12-5 所示。

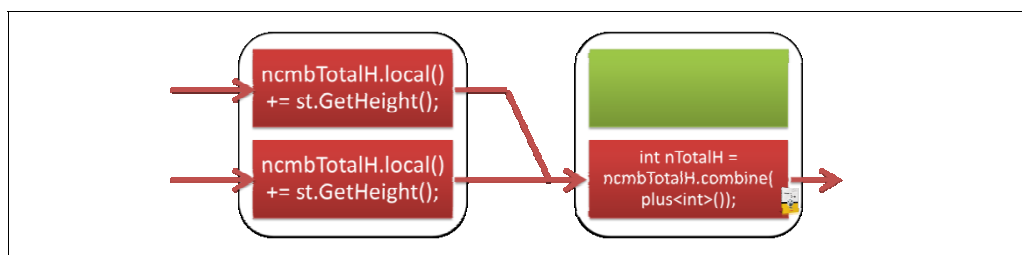


图 12-5 并行对象的执行流程

后记

话说“唐僧”师徒 4 人正前往西天取经，一日夕阳西斜，走得乏了，正想找个客栈打尖休息。还是“悟空”眼尖，远远看见前面有间客栈：“师傅，你看，前面有间客栈。”悟空喊道。

唐僧正骑在白龙马上打瞌睡，听悟空这么一喊，顿时来了精神，抬眼望去，果然看见前面不远处有间客栈，便招呼徒弟们：“悟空、悟能、悟净，还有白龙马，我们今日就在这里休息吧！”（真够啰唆的）说完拍拍马屁股，径直朝客栈前去。白龙马屁股上被八戒烙上去的“BMW”三个大字在夕阳的余晖照射下，甚是醒目。

不一会，唐僧师徒 4 人便到了客栈。小二牵了白龙马到后院去，4 人来到客栈大厅，找个安静的角落入座了，准备要些饭菜吃了好休息，明日趁早赶路。刚入座，悟空便用火眼金睛将整个大厅扫了一番，看看有没有隐藏的“妖怪”。只见大厅七八张桌子，十几个食客，甚是平常。只是堂上多了个说书的，一人，一桌，一椅，一鼠标，再加上个扩音喇叭，在那儿说着什么乔峰、盖茨、燕小六、“瘟到死七”，不知道是哪朝哪代的成年旧事。悟空见这人行有些怪异，恐是妖怪，便拦住店小二打听：“小二，这人是谁啊？在那唠嗑些什么乱七八糟的？”

“客官你说那个说书的啊，据说是个程序员，还是×××（某著名品牌）最有价值的专家。这不，最近金融危机，下岗再就业啦，仗着小时候看过几本武侠小说，死皮赖脸地在这儿摆摊说书了，赶都赶不走，老板见没有坏处，也就由他去了。”小二说完，拿眼瞟了瞟正唾沫横飞的说书人，自己忙去了。

“师傅别担心，只是个说书的。专家？想我东土大唐，文昌武盛，专家论斤卖！”

道：“这位客官可就误会了，其实我是一个程序员！Programmer, understand?”

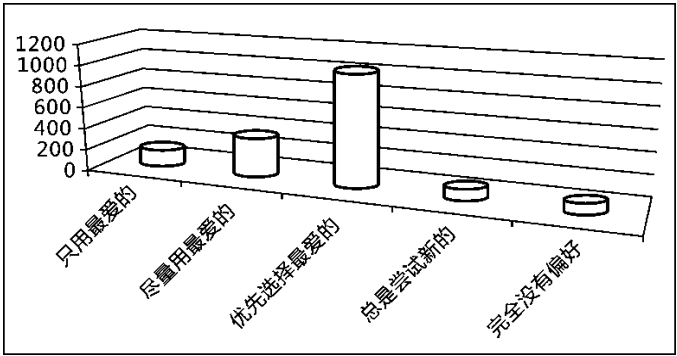
“我想各位听众江湖恩怨也听烦了，不如我来给大家说说软件开发界的奇闻轶事，甚是有趣。”

众食客早就不耐烦了他那几个说了几百遍的段子，都能倒背如流了。听他要改变话题，都纷纷来了兴致，鼓掌叫好，看他要说什么新鲜名堂。

说书的像是受到了鼓舞，把鼠标重重地一拍，说出下面这样一段书来。

话说这 21 世纪，是软件的世纪。至于软件的重要性，自然不需要多说。今个就从软件的起源地说起。软件本起源于美国，后才传入我国。所以说功力深厚，还数人家美国。美国有两个闻名的软件帮派，一曰 CodeGuru，一曰 CodeProject。这两帮派高手如云，一如我中土的武当少林。CodeGuru 先不说，单说这 CodeProject。CodeProject 创立于 1999 年，至今已有 10 余年的历史了。CodeProject 是一个免费公开源代码的程序设计网站，使用者主要是 Windows 平台上的电脑程序设计人员。该网站上的每篇文章几乎都附有源码（src）和例子（demo）供大家下载。经过 10 余年的发展，现在已家喻户晓，成为每个 Windows 程序员必收藏的网站。

CodeProject 的“掌门人”根据近期的热点，每周出一道调查题，各位访客根据自己的情况，用鼠标点点找到适合自己的答案，此之谓“用鼠标说话”。这不，情人节已近，“掌门人”出了这样一道题，如图 1 所示。问各位程序员对自己的情人（也就是自己使用的开发语言）钟情几许？



附录图 1 “你到底爱我有深”大调查

调查结果甚是耐人寻味。

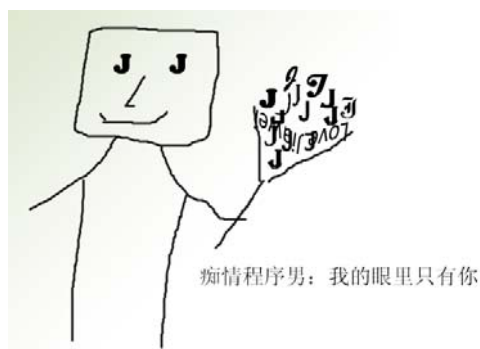
超过 60% 的程序员这样说：“Darling，我爱你，但是有时候我要和别人在一起！”

这部分程序员虽然有自己喜欢的语言，但是不得不使用其他语言工作。

超过 20% 的程序员总是尽量使用自己最喜欢的语言。比起前一种程序员，这种程序员算是比较钟情的了。

最后不到 20% 的程序员，大声地说出了“我的眼里只有你”！如图 2 所示。

这些程序员只用自己的语言工作，只用自己的钟情的语言表达自己的思想。这种人，简直是程序员中的杨过，对自己喜欢的开发语言情有独钟。女生找男朋友，就应该找这样的！



附录图 2 我的眼里只有你

“呵呵呵，师傅，我看这帮程序员，见一个爱一个，哪个好使用哪个，哪个听话用哪个，还没有我老猪专情呢！娶老婆过日子，千万别像程序员。”

八戒说话瓮声瓮气，加上表情逗趣，引来堂下一阵哄笑。众人纷纷附和道：“就是，千万别像程序员。”

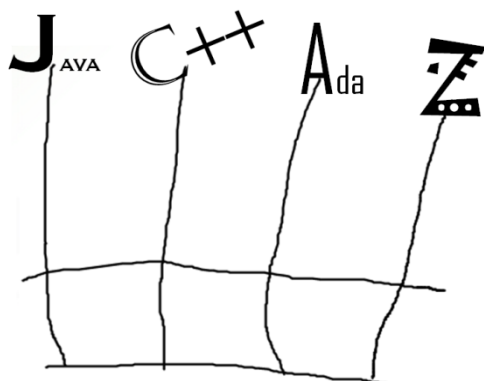
“说书的刚才还说自己是程序员呢，哈哈……”

“别找程序员做男朋友。”

“要找也找那‘我的眼里只有你’的程序员做男朋友。”

说书人没有想到一段说书竟然失了自己的面子，只好收拾起鼠标灰溜溜地走了。回到租住的民房才想起来，这世上到底有多少开发语言啊，这帮哥们见一个爱一个，倒让我今天丢了个大面子。于是翻出压箱底的《金瓶梅》，哦，不对，是维基百科，搜索了一把“程序设计语言”。真是不看不知道，一看吓一跳啊，没有想到世间竟有如此多的程序设计语言，什么带钩的（Java）、带刺的（C++）、带尖的（Ada）、带刃的（Z），要什么样的有什么样的，想想那是比原来公司的美女多了去了，难怪兄弟们见一个爱一个。

但是说书人也发现，在这么多程序设计语言中，有一种叫 C++ 的程序设计语言却受到很多程序员的喜爱，俨然是程序设计语言当中的大众情人。自美国 AT&T 贝尔实验室的本贾尼·斯特劳斯特卢普（Bjarne Stroustrup）博士在 20 世纪 80 年代初期实现了 C++ 之后，在面向对象语言迅速发展的时代背景下，C++ 以其面向对象的语言特性以及对 C 语言的良好兼容，凭借接近 C 语言的效率，在工业界占据了相当多的份额。在以后的发展中，C++ 语言不断引入新的内容，比如标准模板库（STL）和后来的 Boost 等程序库的出现、泛型程序设计的流行，使得 C++ 语言牢牢占据了 TIOBE 编程语言排行榜前三的位置，成为业界最流行的编程语言之一。如果要问程序员们最喜欢什么开发语言，他们大多数都会回答“C++ 语言”，如图 3 所示。



附录图 3 程序员的十八般兵器

说书人不由得感慨了一番，忽然觉得脑子里有点东西，想写下来。于是拿出自己的笔记本（不用电的那种笔记本啊），写下了下面几段话。

60%的程序员是实用主义者，哪管他什么语言，只要能解决问题就行。就像领导人说的，管它黑猫白猫，只要抓到老鼠就是好猫。语言只是工具而已，解决问题才是硬道理。那不到 20%的痴情程序员，是好“老公”的料，却不一定是个好程序员。当然，大师级的人物除外。

世间程序设计语言如此之多，没有高低贵贱之分，只有擅长的领域不同。不同语言，自有它各自擅长的不同领域。常常有人问学习某某语言有没有前途，其实不能问语言有没有前途，需要先问你是否从事什么领域。你所要从事的领域，自有它钟情的语言，学习它就好了。

学好 C++ 语言，走遍天下都不怕！

那个猪头猪脑的和尚甚是讨厌，下回有机会也说他一说！

说书人写完，合上笔记本，顺手拿起书桌上的那本《我的第一本 C++ 书》，与他的 C++ 情人相会去了。